

DISS. ETH NO. 28054

PERFORMANCE MODELING AND OPTIMAL
SCHEDULES FOR DATA OBLIVIOUS
PROGRAMS

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

GRZEGORZ KWASNIEWSKI
Magister Inżynier in Automatics and Robotics
AGH University of Science and Technology

born on the 1st of July 1989
Krakow, Poland

accepted on the recommendation of

Prof. Dr. Torsten Hoefler, examiner
Prof. Dr. Oded Schwartz, co-examiner
Dr. Joost VandeVondele, co-examiner

2021

ABSTRACT

Developing high-performing scientific applications is a challenging task, as it requires a deep understanding of not only algorithmic aspects but also ever-changing hardware characteristics. To guide code development, rigorous performance models are crucial: they reveal underlying mechanisms that determine code behavior, predict performance bottlenecks, and help discover previously unexplored optimization opportunities. In this thesis, we focus on data oblivious programs — programs whose execution trace may be derived solely from the source code and (possibly, symbolic) parameters, but does not depend on the input data. Within this domain, we develop new mathematical tools that capture key aspects that determine programs' performance: arithmetic, parallel, and input/output (I/O) complexities.

To model performance, we begin with the arithmetic and parallel complexities. We model the execution of loop nests as a multidimensional linear system. Our representation allows us to derive non-polynomial loop iteration counts that appear, e.g., in parallel reductions. We then proceed to minimize data movement, a vital optimization step for modern scientific applications. To model the I/O complexity, we define a class of programs called SOAP: Simple Overlap Access Programs that covers a wide range of performance-critical programs, such as linear algebra libraries, stencil computations, and neural networks. Within SOAP, we use the red-blue pebble game abstraction to precisely model data movement, capturing such motifs as recomputation and data reuse across kernels. We show that the SOAP model is both more precise than state-of-the-art methods, improving previous I/O lower bounds for many important kernels, e.g., the Polybench suite, by up to 14 times; and more general, allowing us to establish first I/O lower bounds for entire neural networks.

To show the importance and applicability of data movement minimizations, we apply our methodology to matrix multiplication and matrix factorizations, both LU and Cholesky. We start by deriving their new parallel I/O lower bounds. We then implement our I/O minimizing distributed algorithms and compare them to the state-of-the-art libraries: MKL, SLATE, cuBLAS, CTF, CARMA, CANDMC, and CAPITAL. Our experiments on the Piz Daint supercomputer confirm the clear advantage of our algo-

rithms, providing up to 12.8x speedup for matrix multiplication, 3x for LU factorization, and 1.8x for Cholesky factorization.

In summary, this thesis contributes the SOAP model together with its implementation, which automatically generates I/O lower bounds for input programs written in C or Python. It also provides high-performance distributed implementations of matrix multiplication (COSMA), LU factorization (COnfLUX), and Cholesky factorization (COnfCHOX) with their thorough complexity analysis. All provided tools are available as open-source libraries.

ZUSAMMENFASSUNG

Die Entwicklung leistungsstarker wissenschaftlicher Anwendungen ist eine anspruchsvolle Aufgabe, da sie ein tiefes Verständnis nicht nur algorithmischer Aspekte, sondern auch der sich ständig ändernden Hardwareeigenschaften erfordert. Um die Codeentwicklung zu leiten, sind rigorose Leistungsmodelle von entscheidender Bedeutung: Sie zeigen zugrunde liegende Mechanismen auf, die das Codeverhalten bestimmen, sagen Leistungsengpässe vorher und helfen dabei, bisher unerforschte Optimierungsmöglichkeiten zu entdecken. In dieser Arbeit konzentrieren wir uns auf sogenannte "data-oblivious" Programme — Programme, deren Ablaufverfolgung allein aus dem Quellcode und (möglicherweise symbolischen) Parametern abgeleitet werden kann, aber nicht von den Eingabedaten abhängt. In diesem Bereich entwickeln wir neue mathematische Werkzeuge, die Schlüsselaspekte erfassen, die die Leistung solcher Programmen bestimmen: arithmetische, parallele und Eingabe/Ausgabe-(I/O)-Komplexitäten.

Um die Leistung zu modellieren, beginnen wir mit der arithmetischen und parallelen Komplexität. Wir modellieren die Ausführung verschachtelter Schleifen als mehrdimensionales lineares System. Unsere Darstellung ermöglicht es uns, nicht-polynomielle Schleifeniterationszählungen abzuleiten, die z. B. in parallelen Reduktionen auftreten. Anschließend minimieren wir die Datenbewegung, ein wichtiger Optimierungsschritt für moderne wissenschaftliche Anwendungen. Um die I/O-Komplexität zu modellieren, definieren wir eine Klasse von Programmen namens SOAP: Simple Overlap Access Programs, die eine breite Palette leistungskritischer Programme abdecken, wie z. B. Bibliotheken für lineare Algebra, Stencil-Berechnungen und neuronale Netze. Innerhalb von SOAP verwenden wir die Red-Blue-Pebble-Abstraktion, um die Datenbewegung präzise zu modellieren und Motive wie Neuberechnung und Datenwiederverwendung über Kernel hinweg zu erfassen. Wir zeigen, dass das SOAP-Modell präziser ist als bestehende Methoden und die früheren I/O-Untergrenzen für viele wichtige Kernel, z. B. die Polybench-Suite, um das bis zu 14-fache verbessert; und allgemeiner, ermöglicht es uns, erste I/O-Untergrenzen für ganze neuronale Netze festzulegen.

Um die Bedeutung und Anwendbarkeit von Datenbewegungsminimierungen zu zeigen, wenden wir unsere Methodik auf Matrixmultiplikation und Matrixfaktorisierungen an, sowohl LU als auch Cholesky. Wir begin-

nen mit der Ableitung ihrer neuen unteren Grenzen für parallele E/A. Anschließend implementieren wir unsere I/O-minimierenden verteilten Algorithmen und vergleichen sie mit den State-of-the-Art-Bibliotheken: MKL, SLATE, cuBLAS, CTF, CARMA, CANDMC und CAPITAL. Unsere Experimente auf dem Supercomputer Piz Daint bestätigen den klaren Geschwindigkeitsvorteil unserer Algorithmen, die eine bis zu 12,8-fache Geschwindigkeit für die Matrixmultiplikation, 3x für die LU-Faktorisierung und 1,8x für die Cholesky-Faktorisierung bieten.

Zusammenfassend trägt diese Arbeit das SOAP-Modell zusammen mit seiner Implementierung bei, das automatisch I/O-Untergrenzen für Eingabeprogramme generiert, die in C oder Python geschrieben sind. Es bietet auch leistungsstarke verteilte Implementierungen von Matrixmultiplikation (COSMA), LU-Faktorisierung (CONFLUX) und Cholesky-Faktorisierung (CONFCHOX) mit ihrer gründlichen Komplexitätsanalyse. Alle bereitgestellten Tools sind als Open-Source-Bibliotheken verfügbar.

ACKNOWLEDGEMENTS

First and foremost, I want to thank my supervisor, Professor Torsten Hoefler. I cannot express enough how his influence and expert advice guided me throughout the years. Through his amazing mentorship and tireless help with each and every project not only he secured the success of my publications, but also shaped me as a researcher and as a person.

I would also want to thank other great researchers I worked with. These collaborations gave me the opportunity to participate in fascinating projects that I am proud to be a part of. Numerous people contributed to the work presented in this thesis, among others (in alphabetical order): Maciej Besta, Alexandru Calotoiu, Oliver Fuhrer, André Gaillard, Lukas Gianinazzi, Marko Kabić, Anton Kozhevnikov, Johannes de Fine Licht, Yishai Oltchik, Carlos Osuna, Jens Eirik Saethre, Raffaele Solcà, and Alexandros Nikolaos Ziogas. I especially want to acknowledge Timo Schneider for the countless times he helped me with both the most trivial, and the most ambitious and challenging problems; and Tal Ben-Nun for his awe-inspiring commitment to every task he was involved in. Many of these people not only became great collaborators but also, and maybe, more importantly, good friends.

I greatly appreciate the opportunity to work with the Swiss National Computing Centre (CSCS). The time I spent collaborating with the group led Joost VandeVondele was extremely fruitful and led to one of the major contributions of this thesis. Working with Joost and Marko was a great and joyful experience. Moreover, I wish to express my gratitude for the whole CSCS team for providing computing infrastructure and support crucial for our research.

Last but not least, I would like to thank my parents: Anna and Marek, for their unconditional love and support throughout all the years of my life. It is them who I owe who I am today. I also want to express my deepest gratitude to the rest of my family, my friends, and my girlfriend, Paulina, who were always there for me.

PUBLICATIONS

Publications that form the basis of this thesis:

- Torsten Hoefler and Grzegorz Kwasniewski, **“Automatic Complexity Analysis of Explicitly Parallel Programs”** *in proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA’14)*
- Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffele Solcà, and Torsten Hoefler, **“Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication”** *in proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’19)* **Best Paper Finalist, SC19 Best Student Paper (1/87)**
- Grzegorz Kwasniewski, Tal Ben-Nun, Lukas Gianinazzi, Alexandru Calotoiu, Timo Schneider, Alexandros Nikolaos Ziogas, Maciej Besta, and Torsten Hoefler, **“Pebbles, Graphs, and a Pinch of Combinatorics: Towards Tight I/O Lower Bounds for Statically Analyzable Programs”** *in proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA’2021)*
- Grzegorz Kwasniewski, Marko Kabić, Tal Ben-Nun, Alexandros Nikolaos Ziogas, Jens Eirik Saethre, André Gaillard, Timo Schneider, Maciej Besta, Anton Kozhevnikov, Joost VandeVondele, and Torsten Hoefler, **“On the Parallel I/O Optimality of Linear Algebra Kernels: Near-Optimal Matrix Factorizations”** *in proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’21)*

Additional publications not part of this thesis:

- Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michał Podstawski, Torsten Hoefler, **“SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing”**

in proceedings of the 22nd International Middleware Conference (Middleware '21), Dec. 2021

- Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Marek Konieczny, Onur Mutlu, Torsten Hoefler, **“SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems”**
in proceedings of the 54th IEEE/ACM International Symposium on Microarchitecture (MICRO-54) , Oct. 2021
- Maciej Besta, Zur Vonarburg-Shmaria, Yannick Schaffner, Leonardo Schwarz, Grzegorz Kwasniewski, Lukas Gianinazzi, Jakub Beranek, Kacper Janda, Tobias Holenstein, Sebastian Leisinger, Peter Tatkowski, Esref Ozdemir, Adrian Balla, Marcin Copik, Philipp Lindenberger, Pavel Kalvoda, Marek Konieczny, Onur Mutlu, Torsten Hoefler, **“GraphMineSuite: Enabling High-Performance and Programmable Graph Mining Algorithms with Set Algebra”**
in proceedings of the 47th International Conference on Very Large Data Bases (VLDB'21) , Aug. 2021
- J. de Fine Licht, Grzegorz Kwasniewski, Torsten Hoefler, **“Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis”**
in proceedings of the 28th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)
- Oliver Fuhrer, Tarun Chadha, Torsten Hoefler, Grzegorz Kwasniewski, Xavier Lapillonne, David Leutwyler, Daniel Luethi, Carlos Osuna, Christoph Schaefer, Thomas Schulthess, Hannes Vogt, **“Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0”**
Geoscientific Model Development. Vol 11, Nr. 4, Copernicus Publications, May 2018
- Maxime Martinasso, Grzegorz Kwasniewski, Sadaf R. Alam, Thomas Schulthess, Torsten Hoefler, **“A PCIe Congestion-Aware Performance Model for Densely Populated Accelerator Servers”**

in proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)

- William Tang, Bei Wang, Stephane Ethier, Grzegorz Kwasniewski, Torsten Hoefler, Khaled Z. Ibrahim, Kamesh Madduri, Samuel Williams, Leonid Oliker, Carlos Rosales-Fernandez, Tim Williams,
“Extreme Scale Plasma Turbulence Simulations on Top Supercomputers Worldwide”

in proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)

- Arnamoy Bhattacharyya, Grzegorz Kwasniewski, Torsten Hoefler,
“Using Compiler Techniques to Improve Automatic Performance Modeling”

in proceedings of the 24th International Conference on Parallel Architectures and Compilation (PACT'15)

CONTENTS

1	INTRODUCTION	1
1.1	Path to optimality	1
1.2	Optimality metrics	2
1.2.1	Arithmetic complexity	2
1.2.2	Input/output (I/O) complexity	3
1.2.3	Parallel complexity	3
1.2.4	Hardware performance	5
1.2.5	Combining different metrics	5
1.3	Optimality in linear algebra	6
1.3.1	Thesis contributions	7
2	PARALLEL EFFICIENCY AND PRECISE ITERATION COUNTS	11
2.1	Introduction	11
2.1.1	Work and depth and parallel efficiency	12
2.2	Problem description	15
2.3	Sketch of the algorithm	17
2.4	Algorithm description	18
2.4.1	Affine representation of nested loops	19
2.4.2	Starting conditions	20
2.4.3	Counting the number of iterations	21
2.4.4	Correctness of the algorithm	23
2.4.5	Multipath loops	26
2.5	Practical Considerations	26
2.5.1	Extensions for non-affine loops	27
2.6	Case Studies	28
2.6.1	NAS Parallel Benchmarks: EP	29
2.6.2	NAS Parallel Benchmarks: CG	29
2.6.3	NAS Parallel Benchmarks: IS	30
2.6.4	Mantevo Benchmarks: CoMD	31
2.7	Discussion	32
2.8	Related Work	33
2.9	Summary	34
3	I/O OPTIMAL PARALLEL MATRIX MULTIPLICATION	35
3.1	Introduction	35
3.2	Background	39
3.2.1	Machine Model	39

3.2.2	Computation Model	39
3.2.3	Optimization Goals	40
3.2.4	State-of-the-Art MMM Algorithms	40
3.3	COSMA: High-Level Description	41
3.4	Arbitrary CDAGs: Lower Bounds	43
3.4.1	Existing General I/O Lower Bound	45
3.4.2	Generalized I/O Lower Bounds	46
3.5	Tight I/O Lower Bounds for MMM	49
3.5.1	Definitions	49
3.5.2	I/O Optimality of Greedy Schedules	51
3.5.3	Greedy vs Non-greedy Schedules	57
3.6	Optimal Parallel MMM	59
3.6.1	Sequential and Parallel Schedules	59
3.6.2	Parallelization Strategies for MMM	60
3.6.3	I/O Optimal Parallel Schedule	62
3.7	Implementation	65
3.7.1	Processor Grid Optimization	66
3.7.2	Enhanced Communication Pattern	67
3.7.3	Communication-Computation Overlap	67
3.7.4	One-Sided vs Two-Sided Communication	68
3.7.5	Communication Buffer Optimization	68
3.7.6	Blocked Data Layout	68
3.8	Evaluation	69
3.9	Results	71
3.10	Related work	78
3.10.1	General I/O Lower Bounds	78
3.10.2	Shared Memory Optimizations	79
3.10.3	Distributed Memory Optimizations	79
3.11	Summary	80
4	I/O OPTIMAL MATRIX FACTORIZATIONS	81
4.1	Introduction	81
4.2	Background	84
4.2.1	Machine Model	84
4.2.2	Input Programs	85
4.2.3	I/O Complexity and Pebble Games	86
4.3	General Sequential I/O Lower Bounds	88
4.3.1	Iteration vector, iteration domain, access set	89
4.3.2	Finding the I/O Lower Bound	92
4.4	Data Reuse Across Multiple Statements	93

4.4.1	Case I: Input Reuse and Reuse Size	94
4.4.2	Case II: Output Reuse and Access Sizes	95
4.5	General Parallel I/O Lower Bounds	96
4.6	I/O Lower Bounds of Parallel Factorization Algorithms	97
4.6.1	LU Factorization	97
4.6.2	Cholesky Factorization	99
4.7	Near-I/O Optimal Parallel Matrix Factorization Algorithms	100
4.7.1	LU Dependencies and Parallelization	101
4.7.2	LU Computation Routines	101
4.7.3	Pivoting	102
4.7.4	I/O cost of <i>CONF</i> LUX	104
4.7.5	Cholesky Factorization	106
4.8	Implementation	106
4.9	Experimental Evaluation	107
4.10	Results	109
4.11	Related Work	117
4.12	Summary	119
5	PRECISE DATA MOVEMENT MODELLING FOR GENERAL CLASS OF PROGRAMS	121
5.1	Introduction	121
5.2	Background	123
5.2.1	General Approach of Modeling I/O Costs	124
5.2.2	I/O Lower Bounds	124
5.3	Simple Overlap Access Programs	125
5.4	I/O Lower Bounds For Single-Statement SOAP	130
5.4.1	Definitions	130
5.4.2	Bounding SOAP Access Size	131
5.4.3	Input-Output Simple Overlap	134
5.4.4	Bounding Maximal Subcomputation	134
5.4.5	I/O Lower Bounds and Optimal Tiling	137
5.5	Projecting Programs onto SOAP	138
5.5.1	Non-Overlapping Access Sets	138
5.5.2	Equivalent Input-Output Accesses	139
5.5.3	Non-Injective Access Functions	139
5.5.4	SDG Subgraphs	141
5.5.5	SDG I/O Lower Bounds	143
5.5.6	New Lower Bounds	146
5.6	Related Work	146
5.7	Summary	148

6	CONCLUSIONS AND FUTURE WORK	149
6.1	Future Work	151
	BIBLIOGRAPHY	153

INTRODUCTION

1.1 PATH TO OPTIMALITY

Humanity is rarely satisfied with a working solution - we strive towards finding the best solution. Naturally, two fundamental questions arise: *What* is the best solution? And *how* can we achieve it?

In computer science, the pursuit of optimality is at least as old as general-purpose computers themselves: computational models, such as the Turing machine [1] introduced in 1936, establish frameworks and set the rules by which we define what “optimality” actually is. As laid down by Hartmanis and Stearns in their seminal paper from 1965 [2], complexity theory attempts to answer the first fundamental question: what is the best solution for a given problem. In some sense, it answers where *not* to look for it, providing minimum requirements on any valid solution. For example, an optimal n -input comparison-based sorting network cannot have a depth smaller than $\Omega(\log n)$ [3]. The second fundamental question then appears: How can we construct such a network? It is easy to construct a valid sorting network with depth $\mathcal{O}(n)$. Is it possible to do better?

Bridging the gap between the lower bound, imposed by the computational complexity, and the upper bound, emerging from a currently best-known algorithm, requires effort from both sides. On the one hand, advances in theory can improve the lower bound. For example, Plaxton and Suel [4] showed that the optimal sorting network requires at least $\Omega(\log^2 n / \log \log n)$ depth. On the other hand, Batcher [5] presented a solution with $\mathcal{O}(\log^2 n)$ depth. Similar progress can be seen for other algorithmic problems, with the matrix-matrix multiplication (MMM) being a prime example. The lower bound on the number of elementary multiplications for multiplying two $N \times N$ matrices is $\Omega(N^2)$ [6]. A “straightforward” MMM algorithm requires $\mathcal{O}(N^3)$ multiplications. However, this upper bound was improved at least twelve times, beginning with the Strassen algorithm [7] introduced in 1969. As of 2021, the best asymptotic upper bound is given by Alman and Williams [8]. Whether there exists an MMM algorithm that achieves the asymptotic lower bound is still an open question.

However, with all these advancements in both the lower bounds and the algorithm design, another question arises — how *practical* are these results?

In the case of sorting, the solution presented by Batcher is widely used, especially in hardware design [9, 10]. However, in the case of MMM, it turns out that except for the Strassen algorithm, almost none of the asymptotically better algorithms are used due to the prohibitive constant factors, which are hidden behind the asymptotic notation. Ballard et al. [11] presented a practical, high-performance parallel Strassen MMM algorithm that asymptotically matches the communication lower bound of the so-called Strassen-like schemas [12]. Despite its applicability, a classical ($\mathcal{O}(N^3)$) MMM algorithm can still perform better on GPUs on smaller matrices [13].

1.2 OPTIMALITY METRICS

Before asking what the optimal solution is, we first need to define a metric of optimality.

1.2.1 Arithmetic complexity

Arguably, the most natural measure of optimality is the arithmetic complexity, that is, the number of elementary arithmetic operations required to evaluate an algorithm. It is the basis of the classical complexity theory, which defines algorithmic complexity classes, such as P , NP , and $EXPTIME$ [14]. It reflects the fact that the number and the speed of arithmetic operations traditionally dominated the overall runtime.

The definition of an “elementary arithmetic operation” may depend on the context. For most numerical algorithms, it is a single operation such as addition or multiplication performed on elementary words of data. However, even in this context, usually only the “most expensive” operations are counted. For instance, for the matrix-matrix multiplication, both additions and multiplications are performed. However, since multiplying two w -digit numbers requires asymptotically more operations than adding them [15], only the number of elementary multiplications is accounted in the asymptotic complexity of MMM. For combinatorial algorithms, such as sorting or graph analytics, an elementary operation is usually a single comparison of two words.

With the introduction of vector machines and SIMD instructions, the elementary operation may be defined as performing one of the arithmetic or logic instructions on the whole input vector. This, in turn, depends on both the instruction vector width v , as well as the length of the elementary word w . Thus, e.g., reducing the floating-point precision allows for packing

more words into the input vector, reducing the total number of performed vector instructions [16].

1.2.2 *Input/output (I/O) complexity*

Simply counting arithmetic operations does not take into account fundamental aspects of how von Neumann’s architecture [17] works — namely, both the program instructions and the input data are stored in the memory. Despite the increasing popularity of “spatial” architectures [18], the von Neumann and other temporal architectures are still dominant. In this model, a single “useful” operation requires (1) fetching an instruction from memory, (2) decoding it, (3) loading the operands, (4) executing the operation, (5) storing the result. It is clear that only step 4 advances an algorithm, while much traffic in and out of the memory is required. To reduce the pressure on the main memory, both the instructions and the operands may be kept in the processor’s cache or registers. However, the cache capacity is usually much too small to fully prevent expensive memory operations. This phenomenon is called von Neumann’s bottleneck [19] and always has been a major concern in algorithmic design. Firstly, due to the scarce hardware resources of early computers, some algorithms might not execute at all due to the insufficient number of registers [20]. Moreover, even if the execution is possible, Tarjan and Paul [21] showed that adding “a bit” of extra memory (that is, polynomially increasing its size) can reduce the total number of load-compute-store operations exponentially. The first I/O cost model of a machine equipped with a two-level memory system — the red-blue pebble game — was introduced by Hong and Kung in their seminal work [22]. Since then, I/O complexity has become an essential field of algorithmic study [23–25].

1.2.3 *Parallel complexity*

The end of Dennard scaling [26] in the mid-2000s brought a fundamental shift in designing high-performance hardware, which inevitably resulted in changes in algorithm design principles to harness the potential of new-generation machines. Dennard scaling states that, while shrinking a hardware feature, the power density of transistors stays constant, simultaneously reducing voltage and current. This improves performance due to higher transistor counts and higher operating frequency without increasing the effective power consumption. While this stayed true for most of the 20th

century, increasing performance “for free”, that is, without any algorithmic changes, is no longer the case anymore. The laws of physics unavoidably led to the increase of the power density when the transistor manufacturing technology went below 20nm [27]. This led to a rapid multicore architecture evolution [28], where overall performance is increased due to parallelism instead of individual operation’s speedup.

This generates another question about optimality: can an optimal sequential algorithm yield suboptimal performance on a parallel machine? To put it in other words, can an algorithm A, which is slower than algorithm B on a sequential machine, eventually “catch up” and produce a solution faster than B if both A and B are given the same number of parallel processors p ? The answer is yes: e.g., for a Single Source Shortest Path (SSSP) problem, an optimal sequential algorithm is given by Dijkstra [29]. However, it is inherently sequential and cannot efficiently utilize parallel resources. On the other hand, the Bellman-Ford algorithm [30], while *not* being work-efficient — on a sequential machine, it is slower than Dijkstra’s — is trivially parallelizable. If both algorithms are run on the same machine with large enough p , Bellman-Ford will eventually outperform Dijkstra’s.

A classical way to reason about the runtime of parallel algorithms is the work-depth model that represents an evaluation of an algorithm as a directed acyclic graph (DAG) $G = (V, E)$. The vertices correspond to elementary computations, while the edges encode data dependencies. The work-depth model further defines work W equal to the total number of vertices $|V|$, and depth D as the length of the longest path in G . The key observation is that, independently of the number of parallel processors p , the overall runtime is lower-bounded by this longest sequential chain of operations D . Brent’s lemma [31] bounds the overall execution time T_p of G on a machine with p processors by $\frac{W}{p} \leq T_p \leq \frac{W}{p} + D$. Valiant [32] introduced the bulk-synchronous parallel (BSP) computation model to capture the number of synchronization steps required in a parallel machine. Grama et al. [33] formalized the parallel scalability of algorithms with the isoefficiency metric: it measures how fast a problem size must increase with p to maintain a constant parallel efficiency.

1.2.4 Hardware performance

In practice, an efficient implementation may impact measured performance more than an improved algorithm. Modern hardware requires a lot of programming effort to harness the potential of wide vector instructions,

hardware prefetching, efficient data layouts, and spatial locality. An algorithm with lower arithmetic complexity may still perform worse due to, e.g., a random access pattern, deep conditional branches, and misaligned data. Therefore, the efficiency of implementation is often measured by % of achieved peak hardware FLOPs performance. Sometimes, a simple loop permutation may increase performance by more than 10x [34] over a naive code. As we show in Chapter 3, highly tuned linear algebra libraries may reach up to 88% of hardware’s peak performance.

However, the aforementioned end of Dennard scaling has yet another consequence, that is, the increasing gap between the compute cost and data movement cost. This is tightly related to the I/O complexity analysis, in which it is often assumed that computation is “for free” and only the data movement cost is counted. Because of this, not only classically memory-bound applications, such as sorting [23] or stencil computations [35], are limited by the memory or network bandwidth: minimizing data movement can also speed up algorithms that are deemed to be compute-bound, as we demonstrate in Chapters 3 and 4.

1.2.5 *Combining different metrics*

For some problems, it can be proven that no algorithm can be optimal in all metrics. Snir [36] showed that for the n -input parallel prefix computation, one can either achieve the work optimality $W = n - 1$ or the depth optimality $D = \log_2 n$, but not both: $W + D \geq 2n - 2$. Solomonik et al. [37] proved that many linear algebra algorithms cannot be at the same time compute-optimal, I/O-optimal, and latency-optimal.

Similar tradeoffs also apply to hardware-related metrics. In our work on simulating the global climate [38], we observe that while evaluating large stencil programs, it is impossible to achieve peak memory bandwidth without redundant data accesses. To fully saturate the memory bus, long bursts of aligned data accesses are required [39]. However, due to the access pattern imposed by stencil computations that require misaligned, offset accesses, a significant fraction of memory bandwidth might be lost. If one optimizes solely for % of achieved memory bandwidth, it is possible to perform redundant loads to create aligned accesses. However, reporting just the bandwidth would be misleading, as these redundant loads do not speed up actual computation. To capture this tradeoff, between the granularity of the irregular access pattern and the imposed bandwidth, we designed the Memory Usage Efficiency (MUE) metric, defined as follows:

$$\text{MUE} = \text{I/O efficiency} \cdot \text{BW efficiency} = \frac{Q}{D} \cdot \frac{B}{\hat{B}}, \quad (1.1)$$

where Q is the I/O lower bound of an algorithm, D is the actual number of data transfers executed, B is the bandwidth achieved by an implementation, and \hat{B} is the maximum achievable bandwidth. It is designed to compare the efficiency of different algorithms (with a different number of data transfers D) that solve the same general problem, such as large-scale weather simulations.

1.3 OPTIMALITY IN LINEAR ALGEBRA

Linear algebra is a backbone for scientific computing. Most of the time-consuming numerical problems are solved using linear methods, such as matrix multiplications [40], factorizations [41], and general tensor contractions [42]. With its ubiquity and importance on the one hand and precise computation models [25, 43] on the other, linear algebra algorithms are crucial targets for performance modeling and algorithmic design. In this field, we observe constant progress in all metrics discussed in Section 1.2.

- **Arithmetic complexity** can be decreased, for example, by exploiting data sparsity [44], using approximate solutions [45] or by using mixed-precision arithmetic [46], especially in conjunction with modern hardware features such as tensor cores [47]¹.
- Improving **I/O complexity** of linear algebra requires tiling [48, 49] and loop fusion [25, 50].
- **Parallel complexity** involves finding optimal parallel decompositions [51] that minimize communication between parallel processors. This is often done on par with the I/O complexity optimizations: parallel machine models can impose restrictions on the size of the local memory available to each processor, yielding memory-dependent parallel lower bounds and algorithms [11, 52–55].

Here we want to state one of the motivations of this thesis: bridging the gap between the lower and the upper bounds. We note that until recently [56], lower bounds were usually given in the asymptotic notation [37, 57–59], or the constant terms were not tight [51]. However, as illustrated with

¹ in the context of mixed precision arithmetic, the “elementary arithmetic operation” is defined as a single SIMD instruction.

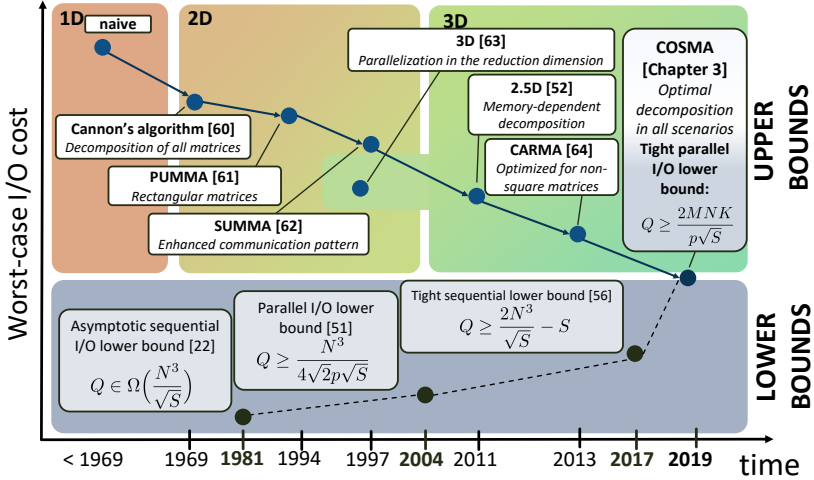


Figure 1.1: Illustrative depiction of the path towards the parallel I/O optimal matrix-matrix multiplication algorithm. Over the 60 years, significant improvement was achieved both in the algorithm design (the upper bound), as well as improving the I/O models (the lower bound). In 2019, we introduced the COSMA algorithm [65], as well as proved the matching parallel I/O lower bound, thus, bridging the gap between the upper and lower bounds.

the example of the MMM algorithms, the asymptotic improvement may be misleading. Especially in the context of linear algebra, where all complexity terms are polynomials with a small degree (usually, up to 3), the constant terms may play a crucial role in determining the final performance of an implementation. We illustrate this in Figure ??: the asymptotic I/O lower bound of MMM was established as early as in 1981 by Hong and Kung [22]. However, this bound was refined multiple items, most notably by Irony et al. [51] and later, by Smith and van de Geijn [56]. The algorithmic improvements span more than 60 years [52, 60–64], with just a single asymptotic improvement from $\mathcal{O}(N^3/\sqrt{p})$, to $\max\{\mathcal{O}(N^3/(p \cdot \sqrt{S})), \mathcal{O}(N^2/(p^{2/3}))\}$, where S is the size of local memory per parallel processor. In Chapter 3, we finally prove the tight parallel I/O lower bound and introduce COSMA — an algorithm that matches this bound for all combinations of parameters p and S , as well as for non-square matrices.

1.3.1 Thesis contributions

The key motivation of this thesis is a design of *provably optimal* and *high-performance* parallel algorithms for scientific computing. In this work, we mainly focus on the parallel and data movement complexities of data oblivious programs: that is, programs whose execution trace depends only on the program parameters (input size, number of processors, size of the fast memory), but not on data contents. Such programs are typically constructed as a series of operations on multidimensional arrays nested within loop nests and constitute a significant fraction of performance-critical scientific applications.

In **Chapter 2**, we establish a mathematical framework to precisely model iteration spaces that extend beyond Presburger arithmetic representation [66]. This allows us to capture the parallel efficiency of algorithms that involve operations such as parallel reductions, possibly yielding logarithmic terms that could not be modeled by classical polyhedral analysis. Our analysis is automated and written as an LLVM [67] pass: given an input program written in C, C++, or Fortran, it automatically generates the work-depth analysis and the code’s parallel efficiency.

In **Chapter 3**, we focus on the parallel I/O complexity. Within the scope of our analysis, we seek to establish *tight* I/O lower bounds. The tightness of the bounds is relevant not only for the theoretical analysis but, most importantly, it provides corresponding I/O optimal schedules. Taking the red-blue pebble game [22] as a starting point, we define the *X*-Partitioning abstraction that precisely captures data reuse. Applying it to the classical MMM algorithm, we obtain the tight parallel I/O lower bound $Q_{MMM} \geq 2MNK/(p \cdot \sqrt{S})$, where M , N , and K are matrix dimensions. The C++ implementation of our algorithm, COSMA, both communicates least and performs best compared to the state-of-the-art libraries: MKL, CARMA, and CTF.

In **Chapter 4**, we generalize our results from Chapter 3 to a class of programs we denote DAAP: Disjoint Array Access Programs. This class covers a wide variety of fundamental scientific kernels, such as matrix factorizations and tensor contractions. The key idea behind DAAP is that by restricting the array access structure, we can precisely count the set sizes required by the *X*-Partitioning abstraction. It allows us to improve the existing sequential I/O lower bound of Cholesky factorization $Q_{Chol} \geq N^3/(3\sqrt{S})$ and generalize it to a parallel machine. We then design parallel LU and Cholesky factorizations algorithms that are within a small factor

from the established lower bounds. Our experiments show that similarly to COSMA, our implementations outperform state-of-the-art solutions: MKL, SLATE, CANDMC, and CAPITAL.

Finally, in **Chapter 5**, we broaden our I/O analysis to capture programs with stencil-like access patterns. Such kernels are ubiquitous not only in weather models or finite-difference methods but also form the basis of convolution operations – crucial components of many machine learning algorithms. Furthermore, with our Symbolic Directed Graph (SDG) abstraction, we explicitly capture data flow in large programs containing multiple kernels, allowing for computation motifs such as recomputation and input and output data reuse. We use it to establish first I/O lower bounds for entire neural networks, such as LeNet-5.

In summary, we believe that this thesis contributes both to a better understanding of fundamental concepts of data oblivious programs, such as work-depth analysis, parallel efficiency, and I/O complexity, as well as practical principles in designing high-performance parallel algorithms. COSMA (Chapter 3) is already integrated into CP2K [68] — a popular quantum chemistry software package developed by the Swiss National Supercomputing Centre, with ongoing work on integrating our matrix factorization libraries, *CONfLUX* and *CONfCHOX* (Chapter 4). The SOAP I/O analyzer (Chapter 5), written in Python, is also being integrated into the DaCe data-centric framework [69], aiming to automate the process of generating provably I/O optimal parallel code.

PARALLEL EFFICIENCY AND PRECISE ITERATION COUNTS

2.1 INTRODUCTION

Parallelism in today's computers is still growing exponentially, currently doubling approximately every two years. This implies that programmers need to expose exponentially growing parallelism to exploit the full potential of the architecture. Parallel programming is generally hard and practical implementations may not always expose enough parallelism to be considered future-proof. This is exaggerated by continuous application development and the fact that applications are developed on systems with significantly lower core counts than their production environment. Thus, it is increasingly important that programmers understand bounds on the scalability of their implementation.

Parallel codes are manifold and numerous programming frameworks exist to implement parallel versions of sequential codes. We define the class of *explicitly parallel* codes as applications that statically divide their workload into several pieces which are processed in parallel. Explicitly parallel codes are the most prevalent programming style in large-scale parallelism using the Pthreads, OpenMP, the Message Passing Interface (MPI), Partitioned Global Address Space (PGAS), or Compute Unified Device Architecture (CUDA) APIs. Many high-level parallel frameworks (e.g., [70]) and domain-specific languages (e.g., [71]) compile to such explicitly parallel languages.

The work and depth model is a simple and effective model for parallel computation. It models computations as vertices and data dependencies as edges of a directed acyclic graph (DAG). The total number of vertices in the graph is the *total work* W and the length of the longest path is called the *depth* D (sometimes also called span). We will now describe more properties of the model and possible analyses.

2.1.1 Work and depth and parallel efficiency

In practice, analyses are often not used to predict the exact running times of an implementation on a particular architecture. Instead, they often determine how the running time behaves with regard to the input size. The work and depth model links sequential running time and parallelism elegantly. The work W is proportional to the time T_1 required to compute the problem on a single core. The depth D is the longest sequential chain and thus proportional to a lower bound to the time T_∞ required to compute the problem with an infinite number of cores.

Work and depth models are often used to develop parallel algorithms (e.g., [72]) or to describe their properties (e.g., in textbooks [73, 74]). Those algorithms are then often adapted in practical settings. We propose to use the same model, somewhat in the inverse direction, to analyze existing applications for bounds on their scalability and available parallelism. Our results can also be used to prove an implementation asymptotically optimal with regards to its parallel efficiency if bounds on work and depth of the problem are known. In our analysis, we use the assumption from [31] that all operations are performed in unit time and the time required for accessing data, storing results, etc., is ignored.

Brent's lemma [31] bounds running times on p cores with $\frac{W}{p} \leq T_p \leq \frac{W}{p} + D$. D measures the sequential parts of the calculation and is equivalent to time t needed to perform an operation with sufficient number of processes, W is equivalent to the number of operations q in Brent's notation and $B = \frac{D}{W}$ is a lower bound of the sequential fraction that limits the returns from adding more cores. Applying Amdahl's law [75] shows that the speedup is limited to $S_p = \frac{T_1}{T_p} \leq \frac{1}{B + \frac{1-B}{p}}$. If we consider the parallel efficiency $E_p = \frac{S_p}{p}$, then we can bound the maximally achievable efficiency using the work and depth model as $E_p = \frac{T_1}{pT_p} \leq \frac{1}{1+B(p-1)}$. We observe that for fixed B , $\lim_{p \rightarrow \infty} E_p = 0$ such that every fixed-size computation can only utilize a limited number of cores *efficiently*, i.e., $E_p \geq 1 - \epsilon$.

This observation allows us to define *available parallelism* and good scaling in terms of ϵ as the maximum number of processes p for which T_p may decrease. Bounds on work and depth for certain problems also allow us to differentiate between a problem that is hard to parallelize (e.g., depth first search (DFS)) and a suboptimal parallelization; we can also define the *distance* of a given parallel code to a *parallelism-optimal* solution.

Work and depth are typically functions of the input size. In structured programming [76], loops and recursion are the only techniques to increase

the work depending on program input parameters. Here, we focus on loops only and we assume that each program can be abstracted as a set of loops that determine the number of executions for each statement. We model each statement as a work item that takes unit time. To simplify the explanation further, we also assume that there is only one statement in each loop (since all statements will have identical iteration counts). Now, the problem of determining the work is equivalent to determine the loop iteration counts for each statement. The depth is relative to a special parameter p that represents the number of processes. We now discuss a simple motivating example:

EXAMPLE I: PARALLEL SORTING SKELETON Assume the following loop is executed by $p > 0$ processes¹ (p equally divides n , $n > 0$ and n is a power of 2):

```

1  for(x=0; x<n/p; x++)
2  for(y=1; y<n; y*=2) S1;

```

All variables that are not changed in the loop but influence the iteration counts are called *parameters*. The parameter n represents the size of the input problem. S1 is an arbitrary computation statement that models one work item. We now analyze work and depth for this explicitly parallel loop.

For any loop, the elements that determine the number of iterations can be split into three classes:

1. Initial assignment: $x=0, y=1$
2. Loop guards: $x<n/p, y<n$
3. Loop updates: $x++, y*=2$

The number of iterations of statement `s1` in this loop (depending on the parameters n and p) can be counted as

$$N(n, p) = T_p = n/p \cdot \log_2(n).$$

From $N(n, p)$, we can determine that the total work and depth is

$$W(n) = N(n, 1) = T_1 = n \cdot \log_2(n)$$

$$D(n) = N(n, \infty) = T_\infty = \log_2(n).$$

The parallelization is work-conserving and the parallel efficiency $E_p = 1$. If this loop implements parallel sorting, then our analysis shows that it is

¹ We use typewriter font to denote source code variables

asymptotically optimal in work and depth [77], and thus exposes maximum parallelism. In this paper, we will show how to perform this analysis automatically.

EXAMPLE II: PARALLEL REDUCTIONS Our second example illustrates a common problem in parallel shared memory codes: reductions. Programmers often employ inefficient algorithms because efficient tree-based schemes are significantly harder to implement. A sequential reduction would be implemented as follows (addition operations on the variable `sum` are performed atomically):

```
1 sum=0; for(i=0; i<n; i++) sum=sum+a[i];
```

A simple parallelization (assuming $n > p$) would be

```
1 for(i=id*n/p; i<min((id+1)*n/p,n); i++)
2     s[id]+=a[i];
3 for(i=0; i<p; i++) sum=sum+s[i];
```

where `id` is the thread number and `s` is an array of size p for keeping the partial sums of each thread. The total number of iterations of the most loaded process is $N(n, p) = T_p = \lceil n/p \rceil + p$ and the efficiency $E_p = (n+1)/(p \lceil n/p \rceil + p^2)$. This implementation is not work-efficient because the lower bound is $T_p = \Omega(n/p + \log_2(p))$ and the efficiency decreases with p^2 . The lower bound can be achieved if we combine partial results of the sum in a tree structure

```
1 for(i=id*n/p; i<min((id+1)*n/p,n); i++)
2     s[id]+=a[i];
3 for(i=1; i<p; i*=2) combine_partial_sums(s);
```

with the iteration count of the most loaded process $N(n, p) = T_p = \lceil n/p \rceil + \lceil \log_2(p) \rceil$. The work of this solution is $W(n) = T_1 = n$ and the depth $D(n) = T_\infty = \infty$ because the parallelization is not work-conserving (more work is created as threads are added). The parallel efficiency is $E_p = n/(p \lceil n/p \rceil + p \lceil \log_2(p) \rceil)$ which decreases slowly because $\log_2(p)$ work is added per process. From E_p , we can derive that the available parallelism is n .

This example shows that it is crucial to catch loop behavior in the analysis of parallel programs. Different implementations solving the same problem may have different work and depth, some of which resulting in limited scalability. Experiments at a small scale may not expose those limitations as the constants are often rather small. However, our analysis enables us to find those issues early during the development.

The main contributions of this Chapter are:

- We develop a mechanism to symbolically bound the number of iterations in program loops depending on the input parameters and the number of processes.
- We show how to interpret the iteration counts in terms of work and depth. This allows the user to determine the parallel efficiency of a given code.
- We briefly outline how our method can be implemented in a compiler or code analysis tool.
- We demonstrate the applicability of our method and analyze a set of real-world applications for their parallel work and depth and efficiencies.

2.2 PROBLEM DESCRIPTION

Counting numbers of loop iterations of arbitrary codes is impossible because even termination of arbitrary loop nests cannot be decided [1]. In our work, we focus on the class of loops where all loop update functions and loop guards are affine functions of *iteration variables*, i.e., variables that change during loop execution. It was shown in previous works that a subset of this class covers many important codes in parallel computing [43].

Our method is strictly more powerful than other iteration counting approaches (e.g., [78]) that require that loop update functions are valid expressions in Presburger arithmetic (which supports only addition and subtraction of symbolic values and constants). We refer the reader to Section 2.8 for a more detailed differentiation. In this paper, we focus on the extraction of work and depth for affine loop nests. To do so, we need to find the number of iterations of the program as a function of the number of processes.

Affine loop. Let $x \in \mathbb{Z}^m$ be an integer-valued *iteration variable vector* and x_0 its *initial assignment* right before entering the loop. We call a loop *affine* if we can present it in the form² :

```

1   $x \leftarrow x_0$            // Initial assignment
2  while( $c^T x < g$ )       // Loop guard
```

² We use an arrow (\leftarrow) symbol to denote an assignment in math notation

```
3   x ← Ax + b      // Loop update
```

Listing 2.1: Affine Loop

The *loop guard* $c^T x < g$ is determined by the constant vector $c \in \mathbb{R}^m$ and bounded by a scalar constant g . The loop update function $Ax + b$, consisting of a real matrix $A \in \mathbb{R}^{m \times m}$ and a constant vector $b \in \mathbb{R}^m$, determines how the iteration variables are updated during each iteration. Each constant may represent a symbolic loop parameter.

Perfectly Nested Loops. We extend our definition to a program consisting of r nested affine loops:

1. Each *loop guard* $c_k^T x < g_k$ at level k is an affine predicate of the iteration variables from levels $1 \dots k$.
2. Each loop body at levels $1 \dots r - 1$ consists of three elements:
 - a) initial assignment - $A_k x + b_k$
 - b) nested loop(s)
 - c) loop update - $U_k x + v_k$

We require well-structured programs [76]: For a loop at level k , the initial assignment, loop guard and loop update may only use variables defined at the same or higher levels $1 \dots k$. Iteration variables of any parent loop at level $1 \dots k - 1$ may not be changed in nested loops at levels $k \dots r$. Such loops can thus be expressed in the general form

```
1  while( $c_1^T x < g_1$ ) {
2    x ←  $A_1 x + b_1$ 
3    while( $c_2^T x < g_2$ ) {
4      ...
5      x ←  $A_{k-1} x + b_{k-1}$ 
6      while( $c_k^T x < g_k$ ) {
7        x ←  $A_k x + b_k$ 
8        while( $c_{k+1}^T x < g_{k+1}$ ) {...}
9        x ←  $U_k x + v_k$  }
10     x ←  $U_{k-1} x + v_{k-1}$  }
11     ...
12    x ←  $U_1 x + v_1$ 
13  }
```

where $A_k, U_k \in \mathbb{R}^{m \times m}$, $b_k, v_k, c_k \in \mathbb{R}^m$, $g_k \in \mathbb{R}$ and $k = 1 \dots r$. Furthermore, $\forall i < k, i \neq j : A_{k,i,j} = U_{k,i,j} = 0$, $\forall i < k, i = j : A_{k,i,j} = U_{k,i,j} = 1$ and $\forall i > k : g_{k,i} = 0$.

Note that even though all the assignments and loop guards are affine, the number of iterations of such a nested loop may not be affine. For example the following affine loop will iterate $\lceil \log_2(n) \rceil$ times:


```

1  x=1;
2  while(x<n) x=2*x;

```

Perfectly nested loops are rare and loops often contain multiple loops at the same level. We now outline how our scheme also supports multipath loops.

Multipath Loops are loops that may contain multiple nested loops in one parent loop body. The example in Listing 2.2 shows such a loop: Inside the outer loop body we have two inner loops. How multiple loops are combined to fit the model description is covered in Section 2.4.5.

```

1  x=1;
2  while(x<n/p+1) {
3    y=x;
4    while(y<m) {S1; y=2*y;}
5    z=x;
6    while(z<m) {S2; z=z+x;}
7    x=2*x;
8  }

```

Listing 2.2: Complex Multipath Loop Nest

It is time-consuming and error-prone for humans to derive work and depth of complex loops like the one shown in Listing 2.2. Our algorithm computes work and depth for each statement automatically. For example, the number of executions N of the statement s_2 is bounded by

$$2^m \left(1 - \left\lceil \frac{n}{p} + 1 \right\rceil^{-1} \right) - \log_2 \left(\left\lceil \frac{n}{p} + 1 \right\rceil \right) \leq N \leq m \left(2 - \left\lceil \frac{n}{p} + 1 \right\rceil^{-1} \right).$$

This bounds the work W on a single process

$$2^m \left(1 - (n+1)^{-1} \right) - \log_2(n+1) \leq W \leq m \left(2 - (n+1)^{-1} \right)$$

and the depth D

$$0 \leq D \leq m.$$

2.3 SKETCH OF THE ALGORITHM

We first introduce the concept of a closed-form *affine representation*. The affine representation of a single affine loop consists of two elements:

1. A single affine statement, which represents the value of the vector x after i iterations of the loop

$$x(i) = L(i) \cdot x_0 + p(i), \text{ and}$$

2. the *counting function* $n(x_0)$ that states how many times the loop will iterate before the loop guard $c^T x(i) < g$ is violated.

The variable i represents the current iteration step. We will refer to i as the *iteration counter*; x_0 is the value of the vector x before entering the loop.

We now provide an intuitive sketch of our algorithm: Given r perfectly nested affine loops, starting from the inner loop, we replace each loop with its affine representation. For r nested loops the result is

$$x(i_1, \dots, i_r) = A_{final}(i_1, \dots, i_r)x_0 + b_{final}(i_1, \dots, i_r) \quad (2.1)$$

where $i_k = 0 \dots n_k(x_{0,k})$, matrix A_{final} and vector b_{final} are the compositions of all L_k and p_k , $k = 1 \dots r$.

The function $n_k(x_{0,k})$ represents the number of iterations in the k th loop with the starting conditions $x_{0,k}$. The starting conditions depend on iteration counters of all the loops at higher levels, i.e., $x_{0,k} = \phi_k(i_1, \dots, i_{k-1})$.

Number of iterations. We can compute the total number of iterations of the innermost loop using the counting function of each loop:

$$N = \sum_{i_1=0}^{n_1(x_{0,1})} \sum_{i_2=0}^{n_2(x_{0,2})} \dots \sum_{i_{r-1}=0}^{n_{r-1}(x_{0,r-1})} n_r(x_{0,r}). \quad (2.2)$$

To solve Equation (2.2), we need to compute:

1. the affine representations for all the loops together with their counting functions $n_k(x_{0,k})$,
2. the starting conditions for all loops as functions of iteration counters $x_{0,k} = \phi_k(i_1, \dots, i_{k-1})$, and
3. all the sums in Equation (2.2).

Work and depth analysis. The number of processes in explicitly parallel programs is always available as a special variable which we call p . In parallelized codes, p is used in loop guards or loop update functions to divide the work into p pieces. Our algorithm determines the number of iterations as a function of all program parameters. We can then define the work of a program as $W = N|_{p=1}$ and depth $D = N|_{p \rightarrow \infty}$. Parallel efficiency and exposed parallelism can be computed as described in Section 2.1.1.

2.4 ALGORITHM DESCRIPTION

We now describe all the steps and approximations needed to solve Equation (2.2) which determines the final loop count.

2.4.1 Affine representation of nested loops

We now explain how we transform a perfectly nested loop into a single affine statement. This statement can then be combined with the initial assignments and the original loop update function of the parent loop into a new loop update function that represents the whole loop nest.

Each loop update statement $x \leftarrow Ax + b$ is a recursive formula for the value of the vector x in the current step, given the value in the previous step. The closed-form of that formula for vector x after i iterations and with the starting value x_0 can be written as

$$\hat{x}(i, x_0) = A^i \cdot x_0 + \sum_{j=0}^{i-1} A^j b. \quad (2.3)$$

Using $x(i, x_0)$, we compute the number of iterations d after which the loop guard is not satisfied

$$n(x_0) = \left\lceil \arg \min_d (c^T \cdot x(d, x_0) \geq g) \right\rceil. \quad (2.4)$$

Equation (2.4) defines the *counting function* $n(x_0)$. Let $L = A^i$ and $p = \sum_{j=0}^{i-1} A^j b$ from Equation (2.3). After we have obtained the closed affine form of a loop at level $k + 1$, we can transform the loop nest at level k to

```

1  while( $c_k^T x < g_k$ ) {
2     $x \leftarrow A_k x + b_k$  // Initial assignment ( $x_{0,k+1}$ )
3     $x \leftarrow L_k x + p_k$  // Nested loop (aff. rep.)
4     $x \leftarrow U_k x + v_k$  // Loop update
5  }
```

where $x = L_k x + p_k$ is the closed-form representation of the $(k + 1)$ st loop. Furthermore, the three affine statements can be combined to one

$$x \leftarrow U_k(L_k(A_k x + b_k) + p_k) + v_k. \quad (2.5)$$

We can then use it to form the affine representation of the parent loop. Applying this procedure recursively for all k loop nests will produce the final affine representation $x = A_{final} x_0 + b_{final}$ that expresses the whole loop nest (cf. Equation (2.1)).

EXAMPLE OF AN AFFINE REPRESENTATION The following example illustrates how to transform a loop into its affine representation. Consider the following loop

```

1  y=y0; z=z0;
2  while(y<z) {y+=2; z--;}

```

that we can write in matrix form as

$$x_0 = \begin{pmatrix} y_0 \\ z_0 \end{pmatrix}, \quad x(i+1) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x(i) + \begin{pmatrix} 2 \\ -1 \end{pmatrix}, \quad c = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

and $g = 0$. Using Equation (2.3), we get the *affine representation* of that loop

$$x(d, x_0) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x_0 + \begin{pmatrix} 2d \\ -d \end{pmatrix}.$$

Equation (2.4) results in

$$\left[\arg \min_d \left(\begin{pmatrix} 1 & -1 \end{pmatrix} \cdot \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x_0 + \begin{pmatrix} 2d \\ -d \end{pmatrix} \right) \geq 0 \right) \right],$$

that can be simplified to $\lceil \arg \min_d (z_0 - y_0 \leq 3d) \rceil$. A symbolic solver (e.g., MuPAD [79]) will determine the solution for $\lceil d = (z_0 - y_0)/3 \rceil$, which leads to the counting function $n(x_0) = \lceil (y_0 - z_0)/3 \rceil$.

2.4.2 Starting conditions

The starting conditions $x_{0,k+1}$ for a loop at level $k+1$ are determined by the value of the vector x before entering the loop. For each loop, at depths $k = 1, \dots, r$, let \hat{x}_k denote the corresponding function defined in equation (2.3), giving the i_k th *initial assignment* at level k , for $i_k = 1, \dots, n_k(x_{0,k})$,

$$x_{0,k+1} = A_k \cdot \hat{x}_k(i_k, x_{0,k}) + b_k. \quad (2.6)$$

We can now count the starting conditions recursively until we reach the top level, where $x_{0,1} = x_0$. In general, the starting condition at level k are compositions of affine representations and initial assignments of all the loops from level $1 \dots k-1$, treating all iteration variables i_1, i_2, \dots, i_{k-1} as parameters.

EXAMPLE OF THE STARTING CONDITION We now show a small example to illustrate the computation of the starting conditions for inner loops. Given the two nested loops

```

1  y=y0; z=z0;
2  while(y<m) {
3      z=y;
4      while(z<m) {z++;}
5      y*=2;
6  }
```

Listing 2.3: Nested Loop

The affine representation for the inner loop with iterator i_2 and starting conditions $x_{0,2}(i_1)$ depending on the outer loop

$$x(i_2) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x_{0,2}(i_1) + \begin{pmatrix} 0 \\ i_2 \end{pmatrix}, \quad n_2 = m - \begin{pmatrix} 0 & 1 \end{pmatrix} \cdot x_{0,2}(i_1),$$

and for the outer loop with iterator i_1 and starting conditions x_0

$$x(i_1) = \begin{pmatrix} 2^{i_1} & 0 \\ \frac{2^{i_1}}{2} & 0 \end{pmatrix} x_0 + \begin{pmatrix} 0 \\ i_2 \end{pmatrix}, \quad n_1 = \left\lceil \log_2 \left(\frac{m}{\begin{pmatrix} 1 & 0 \end{pmatrix} \cdot x_0} \right) \right\rceil.$$

The initial assignment for the outer loop is

$$A_1 = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}; \quad b_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \quad x_0 = \begin{pmatrix} y_0 \\ z_0 \end{pmatrix}.$$

Starting conditions for the inner loop using Equation (2.6) are

$$x_{0,2}(i_1) = A_1 x(i_1) + b_1 = \begin{pmatrix} 2^{i_1} & 0 \\ 2^{i_1} & 0 \end{pmatrix} x_0 = \begin{pmatrix} 2^{i_1} y_0 \\ 2^{i_1} y_0 \end{pmatrix}.$$

Using Equation (2.2) leads to the exact number of iterations.

2.4.3 Counting the number of iterations

All counting functions and starting conditions can be combined into a final symbolic iteration count. First, we compute all r starting conditions $x_{0,1}, x_{0,2}, \dots, x_{0,r}$ in the form $x_{0,1} = x_0$, $x_{0,2} = f_2(i_1, x_0)$, \dots , $x_{0,r} = f_r(i_1, i_2, \dots, i_{r-1}, x_0)$. Then, we compute all counting functions in the form $n_1(x_{0,1})$, $n_2(x_{0,2})$, \dots , $n_r(x_{0,r})$. This enables us to calculate the sums of Equation (2.2) and solve for the final loop iteration count and derive work and depth from this. We use a symbolic solver to simplify and solve the equations.

In some cases, it is not possible to symbolically determine the exact solution from Equation (2.2). We differentiate two cases:

1. The counting function contains a ceiling, e.g.,

$$\sum_{i=1}^{\lceil \frac{n}{2} \rceil} i = \begin{cases} \frac{(n+2)n}{8} & \text{if } n \text{ is even} \\ \frac{(n+3)(n+1)}{8} & \text{if } n \text{ is odd} \end{cases}$$

2. The symbolic solver cannot find a closed form, e.g.,

$$\sum_{i=1}^n i \cdot \log_2(i).$$

In both cases, we derive lower and upper bounds of the respective sum.

Bounded Sum Approximation (BSA) Algorithm. We now show our BSA algorithm that tightly approximates lower and upper bounds of Equation (2.2) in the two cases where the exact solution cannot be determined.

Obtaining bounds in the first case is simple. For a function $\lceil f(n) \rceil$, we determine the upper bound as $f(n) + 1$ and the lower bound as $f(n)$.

For the second case, with no symbolic solution for a sum, we approximate the sum with an integral [80]. For a non-decreasing function $f_1(i)$

$$\int_0^{n+1} f_1(x-1) dx \leq \sum_{i=0}^n f_1(i) \leq \int_0^{n+1} f_1(x) dx,$$

and for a non-increasing function $f_2(i)$

$$\int_0^{n+1} f_2(x-1) dx \geq \sum_{i=0}^n f_2(i) \geq \int_0^{n+1} f_2(x) dx.$$

If $f(i)$ is not monotonic in the interval $[0, n]$, then we split it into smaller intervals in which $f(i)$ is monotonic. For this, we compute the first $\frac{df}{di}$ and second $\frac{d^2f}{di^2}$ derivatives symbolically. Then we apply the approximation in each segment and combine them to get the proper upper and lower bounds.

While solving Equation (2.2) we need to carry the lower and upper bounds forward recursively. In the branch of the lower bounds, we only consider lower bounds of parent loops and similarly in the upper bound branch. We may require case differentiations if some counting functions are not monotonic. However, we rarely observed non monotonic counting functions in practice.

EXAMPLE OF BOUNDED SUM APPROXIMATION Assume the following nested loop:

```

1  k=1; l=2;
2  while(k>0) {
3      m = k;
```

```

4  while(m<s) m++;
5  k = k+l;
6  l--;
7  }

```

We see that $k_{0,1} = k_0 = 1$ and $l_{0,1} = l_0 = 2$. The counting function for the inner loop is

$$n_2 = s - k_{0,2}$$

and for the outer loop

$$n_1 = \left\lceil l_{0,1} + \frac{\sqrt{4l_{0,1}^2 + 4l_{0,1} + 8k_{0,1} + 1 + 1}}{2} \right\rceil = 6.$$

The starting conditions for the inner loop are

$$k_{0,2} = k_{0,1} + i_1 \cdot l_{0,1} - \frac{i_1 \cdot (i_1 - 1)}{2} = -\frac{1}{2}i_1^2 + \frac{5}{2}i_1 + 1.$$

The number of iterations of the loop nest, according to Equation 2.2 is

$$N = \sum_{i_1=0}^{n_1} n_2 = \sum_{i_1=0}^{n_1} \left(s + \frac{1}{2}i_1^2 - \frac{5}{2}i_1 - 1 \right)$$

We now approximate this sum with an integral. Analyzing the first and second derivative of n_2 shows that within the interval $(0, n_1)$ the function is decreasing in $(0, 2.5)$ and increasing in $(2.5, n_1)$. The sum can then be bounded from above by

$$U = \int_0^2 n_2(i_1 - 1) di_1 + n_2(2) + \int_3^{n_1} n_2(i_1) di_1 = 6s - \frac{47}{12}$$

and from below by

$$L = \int_0^2 n_2(i_1) di_1 + n_2(2) + \int_3^{n_1} n_2(i_1 - 1) di_1 = 6s - \frac{161}{12}.$$

Figure 2.1 illustrates the example.

2.4.4 Correctness of the algorithm

All the steps of the algorithm except the sum approximation are proper algebraic transformations. If no approximation is needed then the algorithm

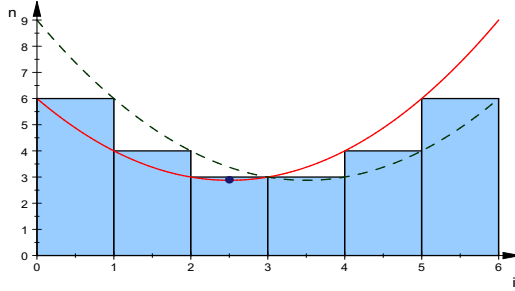


Figure 2.1: Series approximation. Bars represent the series $n_2(i)$, the red line shows the function $n_2(k)$ and the dashed green line shows the function $n_2(k - 1)$. The function $n_2(k)$ over-approximates the series in the interval $[0, 2]$ and under-approximates it in the interval $[3, 6]$; the dot shows the saddle point.

produces the exact number of iterations. For example for the loops in Listing 2.3 the total number of iterations of the inner loop is exactly

$$N = y_0 - 2 \left\lceil \log_2 \left(\frac{m}{y_0} \right) \right\rceil y_0 + m \left\lceil \log_2 \left(\frac{m}{y_0} \right) \right\rceil.$$

If approximation is required, then we need to prove that we obtain proper upper and lower bounds and that this property propagates further through the algorithm.

Lemma 2.1. *The Bounded Sum Approximation algorithm gives valid lower and upper bounds for Equation (2.2).*

Proof. Ceiling upper and lower bound for type 1 approximations are correct by the definition of the ceiling function. Let us consider type 2 sum approximations.

First we need to prove that the upper and lower bounds for a series $f(n)$, where $n = 0, \dots, k$, found by the algorithm are correct. Let's denote

$$U_{[a,b]}(x) = \begin{cases} f(x), & \text{if } \forall x \in [a, b] : \frac{df}{dx} \geq 0 \\ f(x - 1), & \text{if } \forall x \in [a, b] : \frac{df}{dx} \leq 0 \end{cases}$$

$$L_{[a,b]}(x) = \begin{cases} f(x - 1), & \text{if } \forall x \in [a, b] : \frac{df}{dx} \geq 0 \\ f(x), & \text{if } \forall x \in [a, b] : \frac{df}{dx} \leq 0 \end{cases}$$

as upper and lower bounds of the monotonic series f in the interval $[a, b]$, as stated in Section 2.4.3.

Let $c \in [0, k]$ be the only saddle point of function $f(x)$. Intervals with multiple saddle points can be split to smaller intervals where each contains a single saddle point. Then, U and L will change from $f(x - 1)$ to $f(x)$ or from $f(x)$ to $f(x - 1)$ at that point c . The upper bound $U_{[0, \lfloor c \rfloor]} = U_{[0, c]} \neq U_{\lceil c, k \rceil} = U_{\lceil c, k \rceil}$ does not change in the intervals $[0, \lfloor c \rfloor]$ and $\lceil \lceil c \rceil, k \rceil$. We can then bound the value of $f(\lfloor c \rfloor)$ with $U_{[0, c]}(\lfloor c \rfloor)$. Thus,

$$\int_0^{\lfloor c \rfloor} U_{[0, c]}(x) dx \geq \sum_{i=0}^{\lfloor c \rfloor} f(i), \quad \int_{\lceil c \rceil}^k U_{\lceil c, k \rceil}(x) dx \geq \sum_{i=\lceil c \rceil}^k f(i)$$

$$U_{[a, c]}(\lfloor c \rfloor) \geq f(\lfloor c \rfloor).$$

From this follows that the upper bound of the non-monotonic sum of series $\sum_{i=0}^k f(i)$, with saddle point c , can be expressed as:

$$U = \int_0^{\lfloor c \rfloor} U_{[0, c]}(x) dx + U_{[0, c]}(c) + \int_{\lceil c \rceil}^k U_{\lceil c, k \rceil}(x) dx \geq \sum_{i=0}^k f(i).$$

The lower bounds discussion follows a similar reasoning.

We now prove propagation of this property through consecutive sums in Equation 2.2: Let f_k be the k th sum from Equation 2.2, and U_k and L_k upper and lower bounds of f_k . We then can present it as

$$\sum_{i_k=0}^{n_k} n_{k+1}(i_1, \dots, i_{k-1}, x_0) = f_k(i_1, \dots, i_k, x_0)$$

and bound it with

$$L_k(i_1, \dots, i_{k-1}, x_0) \leq f_k(i_1, \dots, i_{k-1}, x_0) \leq U_k(i_1, \dots, i_{k-1}, x_0).$$

The next sum at level $k - 1$ will be

$$\sum_{i_{k-1}=0}^{n_{k-1}} f_k(i_1, \dots, i_{k-1}, x_0)$$

Upper and lower bounds are not changed by summing, such that

$$\sum_{i_{k-1}=0}^{n_{k-1}} f_k(i_1, \dots, i_{k-1}, x_0) \geq \sum_{i_{k-1}=0}^{n_{k-1}} L_k(i_1, \dots, i_{k-1}, x_0)$$

and

$$\sum_{i_{k-1}=0}^{n_{k-1}} f_k(i_1, \dots, i_{k-1}, x_0) \leq \sum_{i_{k-1}=0}^{n_{k-1}} U_k(i_1, \dots, i_{k-1}, x_0)$$

implies $L_1(x_0) \leq f_1(x_0) = N \leq U_1(x_0)$. □

2.4.5 Multipath loops

We formalize a loop containing multiple statements as

```

1  while( $c_k^T x < g_k$ ) {
2     $x \leftarrow A_{k,1}x + b_{k,1}$ 
3     $x \leftarrow A_{k,2}x + b_{k,2}$ 
4    ...
5     $x \leftarrow A_{k,m}x + b_{k,m}$ 
6  }
```

where each of the statements $x \leftarrow A_{k,i}x + b_{k,i}$ may be a simple assignment or an affine representation of a loop. We compose them in the same way as we did in Equation (2.5), forming a single affine statement.

Starting conditions. We compute the starting conditions for each loop by generalizing Equation (2.6). For multipath loops the starting condition for a loop represented by its affine representation $x \leftarrow A_{k,i}x + b_{k,i}$ is the composition of all the affine statements that precede it:

$$x_{0,k+1,i} = A_{k,i-1}(\dots(A_{k,1} \cdot \hat{x}_k(i_k, x_{0,k-1}) + b_{k,1}) \dots) + b_{k,i-1}$$

For illustration consider the following example loop:

```

1  while( $c_k^T x < g_k$ ) {
2     $x \leftarrow A_{k,1}x + b_{k,1}$ 
3     $x \leftarrow A_{k,2}x + b_{k,2}$ 
4     $x \leftarrow A_{k,3}x + b_{k,3}$ 
5     $x \leftarrow A_{k,4}x + b_{k,4}$ 
6     $x \leftarrow A_{k,5}x + b_{k,5}$ 
7  }
```

Assume that in the example above, $x \leftarrow A_{k,2}x + b_{k,2}$, $x \leftarrow A_{k,3}x + b_{k,3}$ and $x \leftarrow A_{k,4}x + b_{k,4}$ are affine representations of three nested loops. Then, the starting condition for the third loop $x \leftarrow A_{k,4}x + b_{k,4}$ is

$$x_{0,k+1,4} = A_{k,3}(A_{k,2}(A_{k,1} \cdot \hat{x}_k(i_k, x_{0,k-1}) + b_{k,1}) + b_{k,2}) + b_{k,3}.$$

Counting the number of iterations. We solve Equation (2.2) using the appropriate counting function $n_r(x_{0,r})$ for each loop. The series of sums is formed according to the hierarchy of loops starting at the innermost loop.

2.5 PRACTICAL CONSIDERATIONS

We now briefly outline how the developed method can be used to assess work and depth of real applications. This paper is focusing on the fundamental techniques, yet, we want to provide a coarse view of how we apply our method in practical settings.

The whole mechanism can be implemented in a source-code analysis tool or a compiler. We use the Low Level Virtual Machine (LLVM [81]) and will outline the implementation. LLVM's internal program representation uses Single Static Assignment (SSA), which makes it simple to determine loops (identified by back-edges), loop guards (identified by conditional branches with back-edges), and all dependent variables.

From this information, we create initial assignments, loop guards, and loop updates for each loop and apply the procedure described in Section 2.4. While the vast majority of loops in practical programs are affine, some loops may depend on more complex conditions and thus do not fit our framework. However, one of the main strengths of our method is that we can still compute the number of iterations of loop nests containing non-affine functions as we will describe in the following section.

2.5.1 Extensions for non-affine loops

If a loop guard is not an affine function of iteration variables and constant parameters then we may not be able to determine the exact number of iterations statically. Examples are loops with iteration counts determined by unknown functions or complex sources like arrays that keep dynamic data. This is often the case in applications that iterate until a complex convergence criterion is reached, e.g., conjugate gradient methods. If the loop exit conditions cannot be represented as affine statements, then the whole block is treated as a symbolic value u (*undefined*).

A strength of our method is that it still solves the remaining affine loop nests symbolically as u is simply treated as a parameter that propagates while solving Equation 2.2. In addition to just treating non-affine loops as new symbolic parameters, our tool enables the user to annotate such loops with affine upper and lower bound functions.

We demonstrate the technique with a loop that we found during one of our case studies. The following Fortran code is extracted from the NAS CG benchmark.

```
1  do j=1,lastrw-firstrow+1
2    sum = 0.d0
3    do k=rowstr(j),rowstr(j+1)-1
4      sum = sum + a(k)*p(colidx(k))
5    enddo
6    w(j) = sum
7  enddo
```

Our tool traces the expression $\text{lastrow} - \text{firstrow} + 1$ back to the program parameter $\text{row_size} = \frac{\text{na}}{\text{nprows}}$ or $\text{row_size} = \frac{\text{na}}{\text{nprows}} + 1$, depending on the process id. This reflects the fact that nprows may not divide na , where na is the problem size. However, the value of $\text{rowstr}(j)$ cannot be determined statically because it represents the location of the first nonzero value in row j of one of the program arrays. Our algorithm then represents the previous loop nest as:

```

1  j=1;
2  while(j<=row_size) {u; j++;}

```

u is treated as a loop with the fixed number of iterations $u = \text{irowstr}(j+1) - \text{rowstr}(j)$. The total number of iterations of this code fragment for the most busy process is represented as

$$N = \left\lceil \frac{\text{na}}{\text{nprows}} \right\rceil u.$$

It is possible to provide the user with information about the exact code fragment that is the origin of u . Users can then determine upper and lower bounds for each unknown parameter.

2.6 CASE STUDIES

In this section we present our results from analyzing several benchmarks. We compute work and depth of several parallel programs to demonstrate the insight we gained into the bounds on parallel efficiency of those practical codes. Our analysis allows us to make statements about parallel efficiency without studying the problem or implementation.

We analyzed major loops of the NAS parallel benchmarks version 3.2 [82] and the Mantevo micro applications version 2.0 [83]. In all the cases presented, no approximation was needed, so presented results give exact number of iterations (with respect to the introduced constants). If not stated otherwise, in the following benchmarks m represents the problem scale, n is a program parameter to NAS specifying the number of iterations to perform, and p denotes number of processes. In some cases processes are arranged into multiple dimensions. In those cases, p_1, p_2, \dots, p_k represents number of processes in corresponding dimensions and $p = \prod_{i=1}^k p_i$. We also assume that the decomposition is square, i.e., $p_1 \approx p_2 \approx \dots \approx p_k$. For easier work-depth analysis, presented results are simplified by replacing constant terms with auxiliary constants c_i . We use constants instead of asymptotic notation to retain lower-order terms.

2.6.1 NAS Parallel Benchmarks: EP

The NAS EP benchmark represents a typical Monte Carlo simulation and thus performs nearly no communication. Our analysis found that only one out of seven loops could not be resolved due to a conditional **goto** statement, resulting in a single u . The work of EP is

$$N = \left\lceil \frac{2^{m-16} \cdot (u + 2^{16})}{p} \right\rceil.$$

The following listing shows the non-affine loop that determines u after removing all statements that do not influence the iteration count:

```

1  u :   do i=1,100
2         ik =kk/2
3         if (ik .eq. 0) goto 130
4         kk=ik
5         continue

```

A programmer can easily determine that $u \leq 100$, which is negligible compared to 2^{16} . Thus, we can approximate the work using one thread $W = T_1 \approx 2^m$ and depth $D = T_\infty \approx 1$. This shows that the parallelization is work-optimal and the efficiency

$$E_p \approx \frac{2^m}{p \left\lceil \frac{2^m}{p} \right\rceil}.$$

This means that $E_p \approx 1$ if $p \lesssim 2^m$ and $E_p \approx 2^m/p$ if $p \gtrsim 2^m$. We conclude that the maximum available parallelism in EP is 2^m , because N cannot further decrease for $p \gtrsim 2^m$. This does not mean that the code will efficiently execute with 2^m tasks, however, it specifies an upper bound to the speedup. This is an expected result since the code is considered “embarrassingly parallel”.

2.6.2 NAS Parallel Benchmarks: CG

The NAS CG program represents a typical conjugate gradient solver. Our tool found that only 2 out of 23 analyzed loops were not affine (cf. Section 2.5.1). The two undefined loops had identical guards, resulting in a single parameter u that can be bounded as $0 \leq u \leq m$. This allows us to bound the number of iterations

$$N \lesssim n \left(g \left\lceil \frac{m}{p} \right\rceil + (6 + 5g) \sqrt{\left\lceil \frac{m}{p} \right\rceil} + (3g + 4) \log_2(\sqrt{p}) \right)$$

where g is the program parameter `cgitmax`. We can approximate work on one thread $W = T_1 \lesssim (g m + \sqrt{m}(5g + 6))n$. However, CG is not work optimal as the parallel work is monotonically growing. This causes the depth to be $D = T_\infty = \infty$. If we treat the problem size m as constant, then CG's parallel efficiency is

$$E_p = c_1 \left(c_2 p \log_2(\sqrt{p}) + c_3 \left\lceil \frac{m}{p} \right\rceil + c_4 \sqrt{\left\lceil \frac{m}{p} \right\rceil} \right)^{-1}.$$

This means that the work per process increases with $\log_2(\sqrt{p})$ due to a parallel reduction among \sqrt{p} processes. The available parallelism is m .

2.6.3 NAS Parallel Benchmarks: IS

The NAS IS program represents a parallel bucket sort algorithm. In each iteration, all processes perform their local sorting and exchange information. The communication overhead is expected to grow with the number of processes. A total of 5 out of 15 analyzed loops were not affine. Those five loops fall in two classes: the first class iterates over maximum and minimum key value represented as $u_1 = \text{max_key_val} - \text{min_key_val} + 1$; the second class iterates over the buckets after redistribution and is represented as $u_2 = \text{bucket_distrib_ptr2}[k] - \text{bucket_distrib_ptr1}[k]$. Both loop iteration counts depend on the structure of the input and the distribution and can thus not easily be bounded tightly. The total number of iterations of IS is

$$N = n \left(3(b + t) + 2 \left\lceil \frac{m}{p} \right\rceil + p + 2 \cdot u_1 + 2 \cdot u_2 \right)$$

where b is the number of buckets, t is the size of the test array, and m is the number of keys to be sorted. The total work on one thread is

$$W = T_1 = n(3(b + t) + 2m + 2u_1 + 2u_2 + 1).$$

The depth $D = \infty$ because the parallelization is not work efficient which is due to the necessary inter-process communications. The parallel efficiency of IS is $E_p = c_1 / \left(p^2 + c_2 p + c_3 p \left\lceil \frac{m}{p} \right\rceil \right)$, which drops quickly with the

number of processes used. The reason is that each process may need to communicate with each other process. The available parallelism is also m in this case.

2.6.4 Mantevo Benchmarks: CoMD

The Mantevo CoMD benchmark represents a classical molecular dynamics simulation. Eight out of 18 analyzed loops contain non-affine statements. The code distributes atoms to processes. The first class updates atoms in the partitions and u_1 represents the number of iterations of those loops.

```

1  u1: while(i<boxes->nAtoms[iBox]) {
2      int jBox=getBox(atoms->r[i0ff+i]);
3      if (jBox!=iBox) moveAtom(i,iBox,jBox);
4      else ++i;
5  }
```

The second class $u_2 = \text{qsort}(n\text{Atoms}[i\text{Box}])$ is limited by the data sizes to be sorted. The number of iterations of the CoMD Benchmark is

$$N = n \left(g(B+3) \left\lceil \frac{m}{p} \right\rceil + g T \left(\left\lceil \frac{m}{p} \right\rceil u_1 + u_2 \right) + \left\lceil \frac{m}{p B} \right\rceil + 2 \right)$$

where B is the fixed amount of atoms in each box, g is the print rate program parameter, and T is the total number of boxes:

$$T = 2 \left(\frac{\sqrt[3]{m}}{p_1} + 2 \right) \left(\frac{\sqrt[3]{m}}{p_2} + \frac{\sqrt[3]{m}}{p_3} + 2 \right) + \frac{\sqrt[3]{m^2}}{p_2 p_3} + \frac{m}{p_1 p_2 p_3}.$$

If we bound $u_1 < B^2$ and $u_2 < B \log_2(B)$, then we can approximate work and depth: $W \lesssim c_1 m + c_2 m^{2/3} + c_3 m^{1/3} + c_4$, and $D \lesssim n(c_5 + c_6 B(\log_2(B)))$. The implementation is work-optimal and the efficiency $E_p \lesssim c_7 / (p + c_8)$ is decreasing with number of processors, which is the result of sequential parts of the program that cannot be parallelized. The available parallelism is m .

SCALABILITY ANALYSIS We were able to determine bounds for work, depth, parallel efficiency, and available parallelism for several real-world applications. We see that the available parallelism in all investigated applications scales linearly with the input problem size. While this suggests good scaling, we show that for CG and IS communication overheads increase the work with the number of processes. For example, for IS, this overhead grows linearly with the number of used processes.

We were able to perform those analyses by pure code introspection which was guided by our tool without requiring knowledge of the implemented methods or algorithms. If the solved problem is known, then one could even proof optimality in terms of parallel efficiency or available parallelism. However, this is outside the scope of this paper.

2.7 DISCUSSION

We now discuss the limitations of our approach and briefly outline potential additional use-cases.

LIMITATIONS Since our analysis only counts loop iterations and does not account for the exact costs of each loop, we can only provide bounds on the expected execution time on a parallel system. However, those bounds are always asymptotically correct. Since we limit ourselves to the work and depth model, we cannot account for communication or synchronization overheads in real codes. Yet, the bounds we provide are useful to determine the relative behavior of work and depth and allow us to reason about exposed parallelism and parallel efficiency just like the work and depth model.

EXTENDING THE MODELS While outside the scope of this paper, it is simple to extend our work and depth models to account for system parameters such as memory or network latency and bandwidth. Blelloch [84] discusses further options.

MODEL-BASED MAPPING TO HETEROGENEOUS SYSTEMS Having a model for the work and depth of each loop in a program can be useful when the program is to be mapped to future heterogeneous architectures. Those systems will most likely contain Latency Compute Units (LCU, cf. today's CPU cores) and Throughput Compute Unites (TCU, cf. today's accelerators such as GPUs or Xeon Phi). A compiler would need to determine the target architecture for each loop statically. It could use the generated work/depth models to assign code pieces with low parallelism (small W/D) to LCUs and code pieces with larger parallelism (large W/D) to TCUs.

2.8 RELATED WORK

Counting loop iterations and assessing scalability of parallel codes are important research problems. Rodriguez-Carbonell and Kapur [85] find polynomial loop invariants using an algebraic approach. Sharma et al. [86] use a data-driven approach to iteratively guess the correct polynomial loop invariant and then check its correctness, and Matringe et al. [87] generate loop invariants also for non-linear differential systems. Loop invariants can be used to bound loop iteration counts, but the resulting bounds are often not tight.

Multiple research groups use the polyhedral model (PM) to determine the exact number of loop iterations [88–90]. In this case, the number of iterations can be approximated by counting integer points in that polyhedron using a polynomial-time algorithm [78]. The PM is widely used, not only in loop analysis [43]. However, it has a serious limitation - it requires that the loop update function can be expressed in Presburger arithmetic and thus cannot deal with non-constant updates such as $x = 2 * x$. To the best of our knowledge, no previous work handled such cases properly. Methods like the one proposed by Blanc et al. [91] require explicitly that loops cannot include such statements.

Other works utilize dynamic approaches to extrapolate program performance and assess scalability in practical settings. Barnes et al. [92] use regression to linear and logarithmic functions to predict scalability of nearly linear-scaling HPC applications. Calotoiu et al. [93] select a scaling model from a set of predefined candidate functions and fit the parameters with regression. Other works, such as [94, 95] use multi-layer neural networks or statistical techniques to predict scalability.

More complex performance prediction frameworks consider the effect of communication [96, 97] and extrapolate single-node runs [98]. Partial execution [99] can improve those techniques. Other studies provide advice for modeling the general performance [100] and scalability [101] of parallel applications. In addition, many application-specific studies exist but cannot be generalized [102, 103].

We extend previous work significantly in two directions: first, we show a technique that can tightly bound the numbers of iterations of arbitrary affine loop nests and second, we show how this method can be used to assess work and depth of parallel applications.

2.9 SUMMARY

We show a method to symbolically count loop iterations of practical codes in terms of their input sizes. Our method provides either an exact solution or tight upper and lower bounds using bounded sum approximation. It is applicable to affine and non-affine loops. While it can bound all affine loops accurately, it handles non-affine loop counts as a symbolic constant and allows the user to provide lower and upper bounds.

We show how to derive parallel work and depth from the loop count models. Using the work and depth model, we approximate bounds on the parallel efficiency of those codes. This technique allows us to specify upper bounds to scalability of practical parallel codes. In general, our method allows a developer to quickly check how an explicitly parallel code scales with the numbers of processes and input sizes.

We are applying a standard algorithmic analysis technique (measuring work and depth) to real source codes. Our developed techniques pave the way to quickly and automatically assess program scalability and will thus quickly become an important tool for future parallel application development and analysis.

I/O OPTIMAL PARALLEL MATRIX MULTIPLICATION

This chapter is an extended version of our publication at the SC'19 conference [65]. Here we introduce the X-Partitioning abstraction, which serves as a starting point for further analysis in the following chapters. The COSMA C++ library was mainly developed — and is still maintained — by Marko Kabić from CSCS.

3.1 INTRODUCTION

Matrix-matrix multiplication (MMM) is one of the most fundamental building blocks in scientific computing, used in linear algebra algorithms [104–106], (Cholesky and LU decomposition [104], eigenvalue factorization [105], triangular solvers [106]), machine learning [107], graph processing [108–113], computational chemistry [40], and others. Thus, accelerating MMM routines is of great significance for many domains. In this chapter, we focus on minimizing the amount of transferred data in MMM, both across the memory hierarchy (*vertical I/O*) and between processors (*horizontal I/O*, aka “communication”). We also focus only on “classical” MMM algorithms that perform N^3 multiplications and additions, in contrast to Strassen-like routines [7]. We note there exist high-performance implementations of fast matrix multiplications [11, 114, 115]. However, they usually outperform classical algorithms only for large matrices [13, 116].

The path to I/O optimality of MMM algorithms is at least 50 years old. The first parallel MMM algorithm is by Cannon [60], which works for square matrices and square processor decompositions. Subsequent works [117, 118] generalized the MMM algorithm to rectangular matrices, different processor decompositions, and communication patterns. PUMMA [61] package generalized previous works to transposed matrices and different data layouts. SUMMA algorithm [62] further extended it by optimizing the communication, introducing pipelining and communication–computation overlap. This is now a state-of-the-art so-called 2D algorithm (it decomposes processors in a 2D grid) used e.g., in ScaLAPACK library [119].

Agarwal et al. [63] showed that in a presence of extra memory, one can do better and introduces a 3D processor decomposition. McColl and

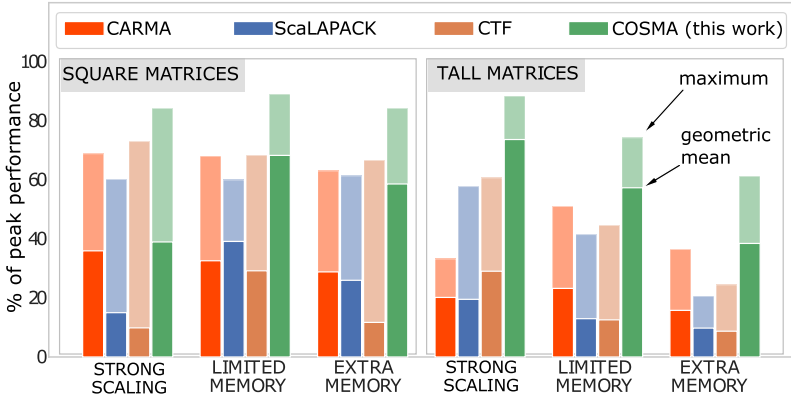


Figure 3.1: Percentage of peak flop/s across the experiments ranging from 109 to 18,432 cores achieved by COSMA and the state-of-the-art libraries (Sec. 3.9).

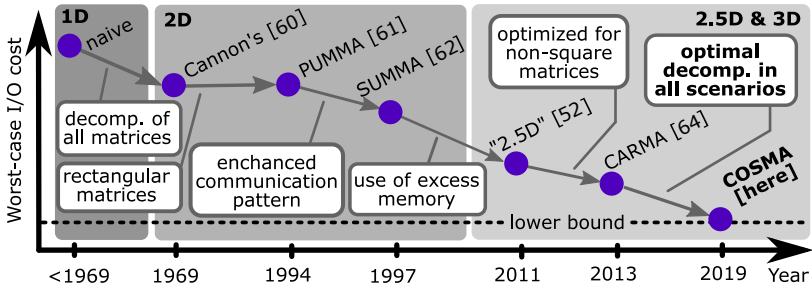


Figure 3.2: Illustratory evolution of MMM algorithms reaching the I/O lower bound.

Tiskin [120] introduced a memory-efficient algorithm in a BSPRAM model with a parametrized tradeoff between communication, synchronization, and memory footprint. The 2.5D algorithm by Solomonik and Demmel [52] uses analogous parametrized decomposition strategy, depending on the available memory. However, Demmel et al. showed that algorithms optimized for square matrices often perform poorly when matrix dimensions vary significantly [64]. Such matrices are common in many relevant areas, for example in machine learning [121, 122] or computational chemistry [123, 124]. They introduced CARMA [64], a recursive algorithm that achieves asymptotic lower bounds for all configurations of dimensions and memory sizes. This evolution for chosen steps is depicted symbolically in Figure 3.2.

Unfortunately, we observe several limitations with state-of-the-art algorithms. ScaLAPACK [119] (an implementation of SUMMA) supports only

	2D [62]	2.5D [52]	recursive [64]	COSMA (this work)
Input:	User-specified grid	Available memory	Available memory, matrix dimensions	Available memory, matrix dimensions
Step 1	Split M and N	Split M, N, K	Split recursively the largest dimension	Find the optimal sequential schedule
Step 2	Map matrices to processor grid	Map matrices to processor grid	Map matrices to recursion tree	Map sequential domain to matrices
		👍 Optimal for	👍 Asymptotically	👍 Optimal for
👎 Requires manual tuning		$M = N$	optimal for all M, N, K, p	all M, N, K
👎 Asymptotically more comm.		👎 Inefficient for $M \ll N$ or $N \ll M$	👎 Up to $\sqrt{3}$ times higher comm. cost	👍 Optimal for all p
		👎 Inefficient for some p	👎 p must be a power of 2	👍👍 Best time-to-solution

Table 3.1: Intuitive comparison between the COSMA algorithm and the state-of-the-art 2D, 2.5D, and recursive decompositions. $C = AB, A \in \mathbb{R}^{M \times K}, B \in \mathbb{R}^{K \times N}$

the 2D decomposition, which is communication-inefficient in the presence of extra memory. Also, it requires a user to fine-tune parameters such as block sizes or a processor grid size. CARMA supports only scenarios when the number of processors is a power of two [64], a serious limitation, as the number of processors is usually determined by the available hardware resources. Cyclops Tensor Framework (CTF) [42] (an implementation of the 2.5D decomposition) can utilize any number of processors, but its decompositions may be far from optimal (Section 3.9). We also emphasize that *asymptotic complexity is an insufficient measure of practical performance*. We later (Section 3.6.2) identify that CARMA performs up to $\sqrt{3}$ more communication. Our observations are summarized in Table 3.1. Their practical implications are shown in Figure 3.1, where we see that all existing algorithms perform poorly for some configurations.

In this chapter, we present COSMA (Communication Optimal S-partition-based Matrix multiplication Algorithm): an algorithm that takes a new approach to multiplying matrices and alleviates the issues above. COSMA is I/O optimal for *all combinations of parameters* (up to the factor of $\sqrt{S}/(\sqrt{S+1}-1)$, where S is the size of the fast memory¹). The driving idea is to develop a general method of deriving I/O optimal schedules

¹ Throughout this dissertation we use the original notation from Hong and Kung to denote the memory size S . In literature, it is also common to use the symbol M [23, 51, 125].

by explicitly modeling data reuse in the red-blue pebble game. We then parallelize the sequential schedule, minimizing the I/O between processors, and derive an optimal domain decomposition. This is in contrast with the other discussed algorithms, which fix the processor grid upfront and then map it to a sequential schedule for each processor. We outline the algorithm in Section 3.3. To prove its optimality, we first provide a new constructive proof of a sequential I/O lower bound (Section 3.5.2.7), then we derive the communication cost of parallelizing the sequential schedule (Section 3.6.2), and finally we construct an I/O optimal parallel schedule (Section 3.6.3). The detailed communication analysis of COSMA, 2D, 2.5D, and recursive decompositions is presented in Table 3.3. Our algorithm reduces the data movement volume by a factor of up to $\sqrt{3} \approx 1.73x$ compared to the asymptotically optimal recursive decomposition and up to $\max\{M, N, K\}$ times compared to the 2D algorithms, like Cannon’s [126] or SUMMA [62].

Our implementation enables transparent integration with the ScaLAPACK data format [127] and delivers near-optimal computation throughput. We later (Section 3.7) show that the schedule naturally expresses communication–computation overlap, enabling even higher speedups using Remote Direct Memory Access (RDMA). Finally, our I/O-optimal approach is generalizable to other linear algebra kernels. We provide the following contributions:

- We propose COSMA: a distributed MMM algorithm that is nearly-optimal (up to the factor of $\sqrt{S}/(\sqrt{S+1}-1)$) for *any combination of input parameters* (Section 3.3).
- Based on the red-blue pebble game abstraction [22], we provide a new method of deriving I/O lower bounds (Lemma 3.4), which may be used to generate optimal schedules (Section 3.4).
- Using Lemma 3.4, we provide a new constructive proof of the sequential MMM I/O lower bound. The proof delivers constant factors tight up to $\sqrt{S}/(\sqrt{S+1}-1)$ (Section 3.5).
- We extend the sequential proof to parallel machines and provide I/O optimal parallel MMM schedule (Section 3.6.3).
- We reduce memory footprint for communication buffers and guarantee minimal local data reshuffling by using a blocked data layout (Section 3.7.6) and a static buffer pre-allocation (Section 3.7.5), providing compatibility with the ScaLAPACK format.

- We evaluate the performance of COSMA, ScaLAPACK, CARMA, and CTF on the CSCS Piz Daint supercomputer for an extensive selection of problem dimensions, memory sizes, and numbers of processors, showing significant I/O reduction and the speedup of up to 8.3 times over the second-fastest algorithm (Section 3.9).

3.2 BACKGROUND

We first describe our machine model (Section 3.2.1) and computation model (Section 3.2.2). We then define our optimization goal: *the I/O cost* (Section 3.2.3).

3.2.1 Machine Model

We model a parallel machine with p processors, each with local memory of size S words. A processor can send and receive from any other processor up to S words at a time. To perform any computation, all operands must reside in processor' local memory. If shared memory is present, then it is assumed that it has infinite capacity. A cost of transferring a word from the shared to the local memory is equal to the cost of transfer between two local memories.

3.2.2 Computation Model

We now briefly specify a model of *general* computation; we use this model to derive the theoretical I/O cost in both the sequential and parallel setting. An execution of an algorithm is modeled with the *computational directed acyclic graph* (CDAG) $G = (V, E)$ [20, 128, 129]. A vertex $v \in V$ represents one elementary operation in the given computation. An edge $(u, v) \in E$ indicates that an operation v depends on the result of u . A set of all immediate predecessors (or successors) of a vertex are its *parents* (or *children*). Two selected subsets $I, O \subset V$ are *inputs* and *outputs*, that is, sets of vertices that have no parents (or no children, respectively).

Red-Blue Pebble Game Hong and Kung's red-blue pebble game [22] models an execution of an algorithm in a two-level memory structure with a small-and-fast as well as large-and-slow memory. A red (or a blue) pebble placed on a vertex of a CDAG denotes that the result of the corresponding elementary computation is inside the fast (or slow) memory. In the initial (or terminal) configuration, only inputs (or outputs) of the CDAG have

blue pebbles. There can be at most S red pebbles used at any given time. A *complete CDAG calculation* is a sequence of moves that lead from the initial to the terminal configuration. One is allowed to: place a red pebble on any vertex with a blue pebble (load), place a blue pebble on any vertex with a red pebble (store), place a red pebble on a vertex whose parents all have red pebbles (compute), remove any pebble, red or blue, from any vertex (free memory). An *I/O optimal complete CDAG calculation* corresponds to a sequence of moves (called *pebbling* of a graph) which minimizes loads and stores. In the MMM context, it is an order in which all N^3 multiplications are performed.

3.2.3 Optimization Goals

Throughout the following chapters, we focus on the *input/output (I/O) cost* of an algorithm. The I/O cost Q is the total number of words transferred during the execution of a schedule. On a sequential or shared memory machine equipped with small-and-fast and slow-and-big memories, these transfers are load and store operations from and to the slow memory (also called the *vertical I/O*). For a distributed machine with a limited memory per node, the transfers are communication operations between the nodes (also called the *horizontal I/O*). A schedule is *I/O optimal* if it minimizes the I/O cost among all schedules of a given CDAG. We also model a *latency cost* L , which is a maximum number of messages sent by any processor.

3.2.4 State-of-the-Art MMM Algorithms

Here we briefly describe strategies of the existing classical MMM algorithms. As stated in the Introduction, we do not consider Strassen-like schemas [11]. Throughout the whole chapter, we consider matrix multiplication $C = AB$, where $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$, $C \in \mathbb{R}^{M \times N}$, where M , N , and K are matrix dimensions. Furthermore, we assume that the size of each matrix element is one word, and that $S < \min\{MN, MK, NK\}$, that is, none of the matrices fits into single processor's fast memory.

We compare our algorithm with the 2D, 2.5D, and recursive decompositions (we select parameters for 2.5D to also cover 3D). We assume a square processor grid $[\sqrt{p}, \sqrt{p}, 1]$ for the 2D variant, analogously to Cannon's algorithm [60], and a cubic grid $[\sqrt{p/c}, \sqrt{p/c}, c]$ for the 2.5D variant [52], where c is the amount of the "extra" memory $c = pS/(MK + NK)$. For the recursive decomposition, we assume that in each recursion level we split

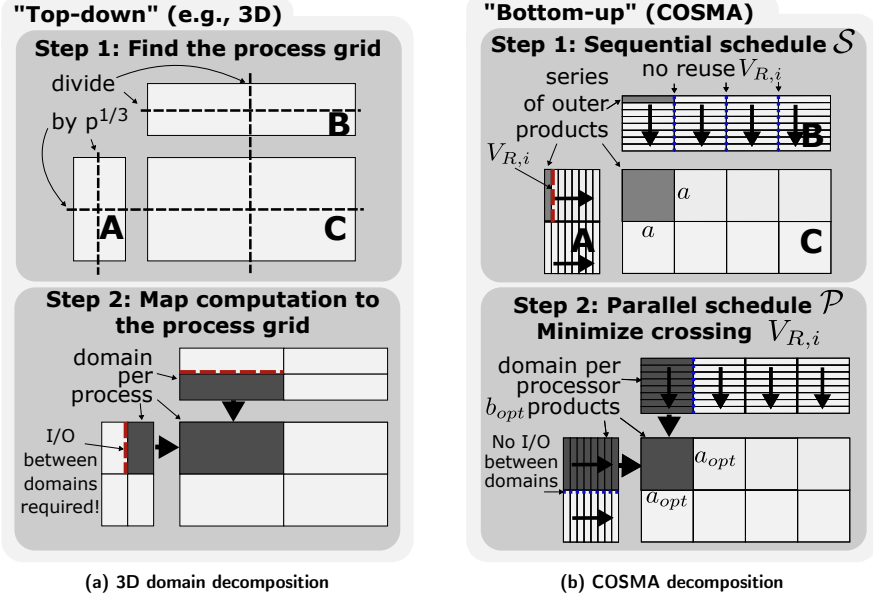


Figure 3.3: Domain decomposition using $p = 8$ processors. In scenario (a), a straightforward 3D decomposition divides every dimension in $p^{1/3} = 2$. In scenario (b), COSMA starts by finding a near optimal sequential schedule and then parallelizes it minimizing crossing data reuse $\mathcal{H}_{R,i}$ (Section 3.5). The total communication volume is reduced by 17% compared to the former strategy.

the largest dimension M, N , or K in half, until the domain per processor fits into memory. The detailed complexity analysis of these decompositions is in Table 3.3. We note that ScaLAPACK or CTF can handle non-square decompositions, however they create different problems, as discussed in Section 3.1. Moreover, in Section 3.9 we compare their performance with COSMA and measure significant speedup in *all* scenarios.

3.3 COSMA: HIGH-LEVEL DESCRIPTION

COSMA decomposes processors by parallelizing the near optimal sequential schedule under constraints: (1) equal work distribution and (2) equal memory size per processor. Such a local sequential schedule is independent of matrix dimensions. Thus, intuitively, instead of dividing a global domain among p processors (the *top-down* approach), we start from deriving a near I/O optimal *sequential* schedule. We then parallelize it, minimizing the I/O

and latency costs Q, L (the *bottom-up* approach); Figure 3.3 presents more details.

Algorithm 1 COSMA

Input: matrices $A \in \mathbb{R}^{M \times K}, B \in \mathbb{R}^{K \times N}$,
 number of processors: p , memory size: S , computation-I/O tradeoff ratio ρ

Output: matrix $C = AB \in \mathbb{R}^{M \times N}$

- 1: $a \leftarrow \text{FindSeqSchedule}(S, M, N, K, p)$ ▷ sequential I/O optimality (Section 3.5)
- 2: $b \leftarrow \text{ParallelizeSched}(a, M, N, K, p)$ ▷ parallel I/O optimality (Section 3.6)
- 3: $(\mathcal{G}, a_{opt}, b_{opt}) \leftarrow \text{FitRanks}(M, N, K, a, b, p, \delta)$
- 4: **for all** $p_i \in \{1 \dots p\}$ **do in parallel**
- 5: $(A_l, B_l, C_l) \leftarrow \text{GetDataDecomp}(A, B, \mathcal{G}, p_i)$
- 6: $s \leftarrow \left\lceil \frac{S - a_{opt}^2}{2a_{opt}} \right\rceil$ ▷ latency-minimizing size of a step (Section 3.6.3)
- 7: $t \leftarrow \left\lceil \frac{b_{opt}}{s} \right\rceil$ ▷ number of steps
- 8: **for** $j \in \{1 \dots t\}$ **do**
- 9: $(A_l, B_l) \leftarrow \text{DistrData}(A_l, B_l, \mathcal{G}, j, p_i)$
- 10: $C_l \leftarrow \text{Multiply}(A_l, B_l, j)$ ▷ compute locally
- 11: **end for**
- 12: $C \leftarrow \text{Reduce}(C_l, \mathcal{G})$ ▷ reduce the partial results
- 13: **end for**

COSMA is sketched in Algorithm 1. In Line 1 we derive a sequential schedule, which consists of series of $a \times a$ outer products. (Figure 3.4 b). In Line 2, each processor is assigned to compute b of these products, forming a *local domain* \mathcal{D} (Figure 3.4 c), that is each \mathcal{D} contains $a \times a \times b$ vertices (multiplications to be performed - the derivation of a and b is presented in Section 3.6.3). In Line 3, we find a processor grid \mathcal{G} that evenly distributes this domain by the matrix dimensions M, N , and K . If the dimensions are not divisible by a or b , this function also evaluates new values of a_{opt} and b_{opt} by fitting the best matching decomposition, possibly not utilizing some processors (Section 3.7.1, Figure 3.4 d-f). The maximal number of idle processors is a tunable parameter δ . In Line 5, we determine the initial decomposition of matrices A, B , and C to the submatrices A_l, B_l, C_l that are local for each processor. COSMA may handle any initial data layout, however, an optimal block-recursive one (Section 3.7.6) may be achieved in

a preprocessing phase. In Line 6, we compute the size of the communication step, that is, how many of b_{opt} outer products assigned to each processor are computed in a single round, minimizing the latency (Section 3.6.3). In Line 7 we compute the number of sequential steps (Lines 8–11) in which every processor: (1) distributes and updates its local data A_l and B_l among the grid \mathcal{G} (Line 9), and (2) multiplies A_l and B_l (Line 10). Finally, the partial results C_l are reduced over \mathcal{G} (Line 12).

I/O Complexity of COSMA Lines 2–7 require no communication (assuming that the parameters M, N, K, p, S are already distributed). The loop in Lines 8–11 executes $\lceil 2ab/(S - a^2) \rceil$ times. In Line 9, each processor receives $|A_l| + |B_l|$ elements. Sending the partial results in Line 12 adds a^2 communicated elements. In Section 3.6.3 we derive the optimal values for a and b , which yield a total of $\min \left\{ S + 2 \cdot \frac{MNK}{p\sqrt{S}}, 3 \left(\frac{MNK}{P} \right)^{2/3} \right\}$ elements communicated.

3.4 ARBITRARY CDAGS: LOWER BOUNDS

We now present a mathematical machinery for deriving I/O lower bounds for general CDAGs. We extend the main lemma by Hong and Kung [22], which provides a method to find an I/O lower bound for a given CDAG. That lemma, however, does not give a tight bound, as it overestimates a *reuse set* size (cf. Lemma 3.3). Our key result here, Lemma 3.4, allows us to derive a constructive proof of a tighter I/O lower bound for a sequential execution of the MMM CDAG (Section 3.5).

The driving idea of both Hong and Kung’s and our approach is to show that some properties of an optimal pebbling of a CDAG (a problem which is PSPACE-complete [130]) can be translated to the properties of a specific partition of the CDAG (a collection of subsets \mathcal{H}_i of the CDAG; these subsets form subcomputations, see Section 3.2.2). One can use the properties of this partition to bound the number of I/O operations of the corresponding pebbling. Hong and Kung use a specific variant of this partition, denoted as S -partition [22].

We first introduce our generalization of S -partition, called X -partition, that is the base of our analysis. We describe symbols used in our analysis in Table 5.1.

X -Partitions. Before we define X -partitions, we first need to define two sets, the *dominator set* and the *minimum set*. Given a subset $\mathcal{H}_i \in V$, define a *dominator set* $Dom(\mathcal{H}_i)$ as a set of vertices in V , such that every path from any input of a CDAG to any vertex in \mathcal{H}_i must contain at least one vertex

MMM	M, N, K	Matrix dimensions
	A, B	Input matrices $A \in \mathbb{R}^{M \times K}$ and $B \in \mathbb{R}^{K \times N}$
	$C = AB$	Output matrix $C \in \mathbb{R}^{M \times N}$
	p	The number of processors
graphs	G	A directed acyclic graph $G = (V, E)$
	$Pred(v)$	A set of immediate predecessors of a vertex v : $Pred(v) = \{u : (u, v) \in E\}$
	$Succ(v)$	A set of immediate successors of a vertex v : $Succ(v) = \{u : (v, u) \in E\}$
I/O complexity	S	The number of red pebbles (size of the fast memory)
	\mathcal{H}_i	An i -th subcomputation of an S -partition
	$Dom(\mathcal{H}_i), Min(\mathcal{H}_i)$	Dominator and minimum sets of subcomputation \mathcal{H}_i
	$\mathcal{H}_{R,i}$	The <i>reuse set</i> : a set of vertices containing red pebbles (just before \mathcal{H}_i starts) and used by \mathcal{H}_i
	$H(S)$	The smallest cardinality of a valid S -partition
	$R(S)$	The maximum size of the reuse set
	Q	The I/O cost of a schedule (a number of I/O operations)
	ρ_i $\rho = \max_i \{\rho_i\}$	The computational intensity of \mathcal{H}_i The maximum computational intensity
Schedules	$\mathcal{S} = \{\mathcal{H}_1, \dots, \mathcal{H}_h\}$	The sequential schedule (an ordered set of \mathcal{H}_i)
	$\mathcal{P} = \{\mathcal{S}_1, \dots, \mathcal{S}_p\}$	The parallel schedule (a set of sequential schedules \mathcal{S}_j)
	$\mathcal{D}_j = \bigcup_{\mathcal{H}_i \in \mathcal{S}_j} \mathcal{H}_i$	The local domain (a set of vertices in \mathcal{S}_j)
	a, b	Sizes of a local domain: $ \mathcal{D}_j = a^2 b$

Table 3.2: The most important symbols used in this chapter.

in $Dom(\mathcal{H}_i)$. Define also the *minimum set* $Min(\mathcal{H}_i)$ as the set of all vertices in \mathcal{H}_i that do not have any children in \mathcal{H}_i .

Now, given a CDAG $G = (V, E)$, let $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_h \in V$ be a series of subcomputations that (1) are pairwise disjoint ($\forall_{i,j,i \neq j} \mathcal{H}_i \cap \mathcal{H}_j = \emptyset$), (2) cover the whole CDAG ($\bigcup_i \mathcal{H}_i = V$), (3) have no cyclic dependencies between them, and (4) their dominator and minimum sets are at most of size X ($\forall_i (|Dom(\mathcal{H}_i)| \leq X \wedge |Min(\mathcal{H}_i)| \leq X)$). These subcomputations \mathcal{H}_i correspond to some execution order (a schedule) of the CDAG, such that at step i , only vertices in \mathcal{H}_i are pebbled. We call this series an X -partition or a *schedule* of the CDAG and denote this schedule with $\mathcal{S}(X) = \{\mathcal{H}_1, \dots, \mathcal{H}_h\}$.

3.4.1 Existing General I/O Lower Bound

Here we need to briefly bring back the original lemma by Hong and Kung, together with an intuition of its proof, as we use a similar method for our Lemma 3.3.

Intuition. The key notion in the existing bound is to use $X = 2S$ -partitions for a given fast memory size S . For any subcomputation \mathcal{H}_i , if $|Dom(\mathcal{H}_i)| = 2S$, then at most S of them could contain a red pebble before \mathcal{H}_i begins. Thus, at least S additional pebbles need to be loaded from the memory. The similar argument goes for $Min(\mathcal{H}_i)$. Therefore, knowing the lower bound on the number of sets \mathcal{H}_i in a valid $2S$ -partition, together with the observation that each \mathcal{H}_i performs at least S I/O operations, we phrase the lemma by Hong and Kung:

Lemma 3.1 ([22]). The minimal number Q of I/O operations for any valid execution of a CDAG of any I/O computation is bounded by

$$Q \geq S \cdot (H(2S) - 1) \quad (3.1)$$

Proof. Assume that we know the optimal *complete calculation* of the CDAG, where a calculation is a sequence of allowed moves in the red-blue pebble game [22]. Divide the complete calculation into h consecutive subcomputations $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_h$, such that during the execution of \mathcal{H}_i , $i < h$, there are exactly S I/O operations, and in \mathcal{H}_h there are at most S operations. Now, for each \mathcal{H}_i , we define two subsets of V , $\mathcal{H}_{R,i}$ and $\mathcal{H}_{B,i}$. $\mathcal{H}_{R,i}$ contains vertices that have red pebbles placed on them just before \mathcal{H}_i begins. $\mathcal{H}_{B,i}$ contains vertices that have blue pebbles placed on them just before \mathcal{H}_i begins, and have red pebbles placed on them during \mathcal{H}_i . Using these definitions, we have: ① $\mathcal{H}_{R,i} \cup \mathcal{H}_{B,i} = Dom(\mathcal{H}_i)$, ② $|\mathcal{H}_{R,i}| \leq S$, ③ $|\mathcal{H}_{B,i}| \leq S$, and ④ $|\mathcal{H}_{R,i} \cup \mathcal{H}_{B,i}| \leq |\mathcal{H}_{R,i}| + |\mathcal{H}_{B,i}| \leq 2S$. We define similar subsets $W_{B,i}$ and $W_{R,i}$ for the minimum set $Min(\mathcal{H}_i)$. $W_{B,i}$ contains all vertices in \mathcal{H}_i that have a blue pebble placed on them during \mathcal{H}_i , and $W_{R,i}$ contains all vertices in \mathcal{H}_i that have a red pebble at the end of \mathcal{H}_i . By the definition of \mathcal{H}_i , $|W_{B,i}| \leq S$, by the constraint on the red pebbles, we have $|W_{R,i}| \leq S$, and by the definition of the minimum set, $Min(\mathcal{H}_i) \subset W_{R,i} \cup W_{B,i}$. Finally, by the definition of S -partition, $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_h$ form a valid $2S$ -partition of the CDAG. \square

3.4.2 Generalized I/O Lower Bounds

3.4.2.1 Data Reuse

A more careful look at sets $\mathcal{H}_{R,i}$, $\mathcal{H}_{B,i}$, $W_{R,i}$, and $W_{B,i}$ allows us to refine the bound on the number of I/O operations on a CDAG. By definition, $\mathcal{H}_{B,i}$ is a set of vertices on which we place a red pebble using the load rule; We call $\mathcal{H}_{B,i}$ a *load set* of \mathcal{H}_i . Furthermore, $W_{B,i}$ contains all the vertices on which we place a blue pebble during the pebbling of \mathcal{H}_i ; We call $W_{B,i}$ a *store set* of \mathcal{H}_i . However, we impose more strict $\mathcal{H}_{R,i}$ and $W_{R,i}$ definitions: $\mathcal{H}_{R,i}$ contains vertices that have red pebbles placed on them just before \mathcal{H}_i begins and – for each such vertex $v \in \mathcal{H}_{R,i}$ – at least one child of v is pebbled during the pebbling of \mathcal{H}_i using the compute rule of the red-blue pebble game. We call $\mathcal{H}_{R,i}$ a *reuse set* of \mathcal{H}_i . Similarly, $W_{R,i}$ contains vertices that have red pebbles placed on them after \mathcal{H}_i ends and were pebbled during \mathcal{H}_i and – for each such vertex $v \in W_{R,i}$ – at least one child of v is pebbled during the pebbling of \mathcal{H}_{i+1} using the compute rule of the red-blue pebble game. We call $W_{R,i}$ a *cache set* of \mathcal{H}_i . Therefore, if Q_i is the number of I/O operations during the subcomputation \mathcal{H}_i , then $Q_i \geq |\mathcal{H}_{B,i}| + |W_{B,i}|$.

We first observe that, given the optimal complete calculation, one can divide this calculation into subcomputations such that each subcomputation \mathcal{H}_i performs an arbitrary number of Y I/O operations. We still have $|\mathcal{H}_{R,i}| \leq S$, $|W_{R,i}| \leq S$, $0 \leq |W_{B,i}|$ (by the definition of the red-blue pebble game rules). Moreover, observe that, because we perform exactly Y I/O operations in each subcomputation, and all the vertices in $\mathcal{H}_{B,i}$ by definition have to be loaded, $|\mathcal{H}_{B,i}| \leq Y$. A similar argument gives $0 \leq |W_{B,i}| \leq Y$.

Denote an upper bound on $|\mathcal{H}_{R,i}|$ and $|W_{R,i}|$ as $R(S)$ ($\forall_i \max\{|\mathcal{H}_{R,i}|, |W_{R,i}|\} \leq R(S) \leq S$). Further, denote a lower bound on $|\mathcal{H}_{B,i}|$ and $|W_{B,i}|$ as $T(S)$ ($\forall_i 0 \leq T(S) \leq \min\{|\mathcal{H}_{B,i}|, |W_{B,i}|\}$). We can use $R(S)$ and $T(S)$ to tighten the bound on Q . We call $R(S)$ a *maximum reuse* and $T(S)$ a *minimum I/O* of a CDAG.

3.4.2.2 Reuse-Based Lemma

We now use the above definitions and observations to *generalize the result of Hong and Kung [22]*.

Lemma 3.2. *An optimal complete calculation of a CDAG $G = (V, E)$, which performs q I/O operations, is associated with an X -partition of G such that*

$$q \geq (X - R(S) + T(S)) \cdot (h - 1)$$

for any value of $X \geq S$, where h is the number of subcomputations in the X -partition, $R(S)$ is the maximum reuse set size, and $T(S)$ is the minimum I/O in the given X -partition.

Proof. We use analogous reasoning as in the original lemma. We associate the optimal pebbling with h consecutive subcomputations $\mathcal{H}_1, \dots, \mathcal{H}_h$ with the difference that each subcomputation \mathcal{H}_i performs $Y = X - R(S) + T(S)$ I/O operations. Within those Y operations, we consider separately $q_{i,s}$ store and $q_{i,l}$ load operations. For each \mathcal{H}_i we have $q_{i,s} + q_{i,l} = Y$, $q_{i,s} \geq T(S)$, and $q_{i,l} \leq Y - T(S) = X - R(S)$.

$$\begin{aligned} \forall_i : |\mathcal{H}_{B,i}| &\leq q_{l,i} \leq Y - T(S) \\ \forall_i : |\mathcal{H}_{R,i}| &\leq q_{s,i} \leq R(S) \leq S \end{aligned}$$

Since $\mathcal{H}_{R,i} \cup \mathcal{H}_{B,i} = \text{Dom}(\mathcal{H}_i)$:

$$\begin{aligned} |\text{Dom}(\mathcal{H}_i)| &\leq |\mathcal{H}_{R,i}| + |\mathcal{H}_{B,i}| \\ |\text{Dom}(\mathcal{H}_i)| &\leq R(S) + Y - T(S) = X \end{aligned}$$

By an analogous construction for store operations, we show that $|\text{Min}(\mathcal{H}_i)| \leq X$. To show that $\mathcal{S}(X) = \{\mathcal{H}_1 \dots \mathcal{H}_h\}$ meets the remaining properties of a valid X -partition $\mathcal{S}(X)$, we use the same reasoning as originally done [22].

Therefore, a complete calculation performing $q > (X - R(S) + T(S)) \cdot (h - 1)$ I/O operations has an associated $\mathcal{S}(X)$, such that $|\mathcal{S}(X)| = h$ (if $q = (X - R(S) + T(S)) \cdot (h - 1)$, then $|\mathcal{S}(X)| = h - 1$). \square

From the previous lemma, we obtain a tighter I/O lower bound.

Lemma 3.3. *Denote $H(X)$ as the minimum number of subcomputations in any valid X -partition of a CDAG $G = (V, E)$, for any $X \geq S$. The minimal number Q of I/O operations for any valid execution of a CDAG $G = (V, E)$ is bounded by*

$$Q \geq (X - R(S) + T(S)) \cdot (H(X) - 1) \quad (3.2)$$

where $R(S)$ is the maximum reuse set size and $T(S)$ is the minimum I/O set size. Moreover, we have

$$H(X) \geq \frac{|V|}{|\mathcal{H}_{max}|} \quad (3.3)$$

where $\mathcal{H}_{max} = \arg \max_{\mathcal{H}_i \in \mathcal{S}(X)} |\mathcal{H}_i|$ is the largest subset of vertices in the CDAG schedule $\mathcal{S}(X) = \{\mathcal{H}_1, \dots, \mathcal{H}_h\}$.

Proof. By definition, $H(X) = \min_{\mathcal{S}(X)} |\mathcal{S}(X)| \leq h$, so $Q \geq (X - R(S) + T(S)) \cdot (H(X) - 1)$ immediately follows from Lemma 3.2.

To prove Eq. (3.3), observe that \mathcal{H}_{max} by definition is the largest subset in the optimal X -partition. As the subsets are disjoint, any other subset covers fewer remaining vertices to be pebbled than \mathcal{H}_{max} . Because there are no cyclic dependencies between subsets, we can order them topologically as $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_{H(X)}$. To ensure that the indices are correct, we also define $\mathcal{H}_0 \equiv \emptyset$. Now, define W_i to be the set of vertices not included in any subset from 1 to i , that is $W_i = V - \bigcup_{j=1}^i \mathcal{H}_j$. Clearly, $W_0 = V$ and $W_{H(X)} = \emptyset$. Then, we have

$$\begin{aligned} \forall_i \quad |\mathcal{H}_i| &\leq |\mathcal{H}_{max}| \\ |W_i| = |W_{i-1}| - |\mathcal{H}_i| &\geq |W_{i-1}| - |\mathcal{H}_{max}| \geq |V| - i|\mathcal{H}_{max}| \\ |W_{H(X)}| = 0 &\geq |V| - H(X) \cdot |\mathcal{H}_{max}| \end{aligned}$$

that is, after $H(X)$ steps, we have $H(X)|\mathcal{H}_{max}| \geq |V|$. \square

From this lemma, we derive the following lemma that we use to prove a tight I/O lower bound for MMM (Theorem 3.1):

Lemma 3.4. *Define the number of computations performed by \mathcal{H}_i for one loaded element as the computational intensity $\rho_i = \frac{|\mathcal{H}_i|}{X - |\mathcal{H}_{R,i}| + |\mathcal{W}_{B,i}|}$ of the subcomputation \mathcal{H}_i . Denote $\rho = \max_i(\rho_i) \leq \frac{|\mathcal{H}_{max}|}{X - R(S) + T(S)}$ to be the maximal computational intensity. Then, the number of I/O operations Q is bounded by $Q \geq |V|/\rho$.*

Proof. Note that the term $H(X) - 1$ in Equation 3.2 emerges from a fact that the last subcomputation may execute less than $Y - R(S) + T(S)$ I/O operations, since $|\mathcal{H}_{H(X)}| \leq |\mathcal{H}_{max}|$. However, because ρ is defined as maximal computational intensity, then performing $|\mathcal{H}_{H(S)}|$ computations requires at least $Q_{H(S)} \geq |\mathcal{H}_{H(S)}|/\rho$. The total number of I/O operations therefore is:

$$Q = \sum_{i=1}^{H(X)} Q_i \geq \sum_{i=1}^{H(X)} \frac{|\mathcal{H}_i|}{\rho} = \frac{|V|}{\rho}$$

\square

3.5 TIGHT I/O LOWER BOUNDS FOR MMM

In this section, we present our main theoretical contribution: a constructive proof of a tight I/O lower bound for classical matrix-matrix multiplication. In Section 3.6, we extend it to the parallel setup (Theorem 3.2). This result is tight (up to diminishing factor $\sqrt{S}/(\sqrt{S}+1-1)$), and therefore may be seen as the last step in the long sequence of improved bounds. Hong and Kung [22] derived an asymptotic bound $\Omega\left(N^3/\sqrt{S}\right)$ for the sequential case. Irony et al. [51] extended the lower bound result to a parallel machine with p processes, each having a fast private memory of size S , proving the $\frac{N^3}{4\sqrt{2p}\sqrt{S}} - S$ lower bound on the communication volume per process. Recently, Smith and van de Gein [56] proved a tight sequential lower bound (up to an additive term) of $2MNK/\sqrt{S} - 2S$. Our proof improves the additive term and extends it to a parallel schedule.

Theorem 3.1 (Sequential matrix multiplication I/O lower bound). *Any pebbling of MMM CDAG which multiplies matrices of sizes $M \times K$ and $K \times N$ by performing MNK multiplications requires a minimum number of $\frac{2MNK}{\sqrt{S}} + MN$ I/O operations.*

The proof of Theorem 3.1 requires Lemmas 3.5 and 3.6, which in turn, require several definitions.

Intuition: Restricting the analysis to greedy schedules provides explicit information of a state of memory (sets $\mathcal{H}_r, \mathcal{H}_{R,r}, \mathcal{W}_{B,r}$), and to a corresponding CDAG pebbling. Additional constraints (Section 3.5.2.7) guarantee feasibility of a derived schedule (and therefore, lower bound tightness).

3.5.1 Definitions

3.5.1.1 Vertices, Projections, and Edges in the MMM CDAG

The set of vertices of MMM CDAG $G = (V, E)$ consists of three subsets $V = \mathcal{A} \cup \mathcal{B} \cup \mathcal{C}$, which correspond to elements in matrices A , B , and MNK partial sums of C . Each vertex v is defined uniquely by a pair (M, T) , where $M \in \{a, b, c\}$ determines to which subset $\mathcal{A}, \mathcal{B}, \mathcal{C}$ vertex v belongs to, and $T \in \mathbb{N}^d$ is a vector of coordinates, $d = 2$ for $M = a \vee b$ and $d = 3$ for $M = c$. E.g., $v = (a, (1, 5)) \in \mathcal{A}$ is a vertex associated with element $(1, 5)$ in matrix A , and $v = (c, (3, 6, 8)) \in \mathcal{C}$ is associated with 8th partial sum of element $(3, 6)$ of matrix C .

For every t_3 th partial update of element (t_1, t_2) in matrix C , and an associated point $v = (c, (t_1, t_2, t_3)) \in \mathcal{C}$ we define $\phi_c(v) = (t_1, t_2)$ to be a projection of this point to matrix C , $\phi_a(v) = (a, (t_1, t_3)) \in \mathcal{A}$ is its projection to matrix A , and $\phi_b(v) = (b, (t_3, t_2)) \in \mathcal{B}$ is its projection to matrix B . Note that while $\phi_a(v), \phi_b(v) \in V$, projection $\phi_c(v) \notin V$ has not any associated point in V . Instead, vertices associated with all K partial updates of an element of C have the same projection $\phi_c(v)$:

$$\begin{aligned} \forall_{v=(c,(p_1,p_2,p_3)),w=(c,(q_1,q_2,q_3)) \in \mathcal{C} : (p_1 = q_1) \wedge (p_2 = q_2)} \\ \iff \phi_c(p) = \phi_c(q) \quad (3.4) \end{aligned}$$

As a consequence, $\phi_c((c, (t_1, t_2, t_3))) = \phi_c((c, (t_1, t_2, t_3 - 1)))$.

A t_3 th update of (t_1, t_2) element in matrix C of a classical MMM is formulated as $C(t_1, t_2, t_3) = C(t_1, t_2, t_3 - 1) + A(t_1, t_3) \cdot B(t_3, t_2)$. Therefore for each $v = (c, (t_1, t_2, t_3)) \in \mathcal{C}$, $t_3 > 1$, we have following edges in the CDAG: $(\phi_a(v), v)$, $(\phi_b(v), v)$, $(c, (t_1, t_2, t_3 - 1), v) \in E$.

3.5.1.2 $\alpha, \beta, \gamma, \Gamma$

For a given subcomputation $\mathcal{H}_r \subseteq \mathcal{C}$, we denote its projection to matrix A as $\alpha_r = \phi_a(\mathcal{H}_r) = \{v : v = \phi_a(c), c \in \mathcal{H}_r\}$, its projection to matrix B as $\beta_r = \phi_b(\mathcal{H}_r)$, and its projection to matrix C as $\gamma_r = \phi_c(\mathcal{H}_r)$. We further define $\Gamma_r \subset \mathcal{C}$ as a set of all vertices in \mathcal{C} that have a child in \mathcal{H}_r . The sets α, β, Γ therefore correspond to the inputs of \mathcal{H}_r that belong to matrices A, B , and previous partial results of C , respectively. These inputs form a minimal dominator set of \mathcal{H}_r :

$$\text{Dom}(\mathcal{H}_r) = \alpha_r \cup \beta_r \cup \Gamma_r \quad (3.5)$$

Because $\text{Min}(\mathcal{H}_r) \subset \mathcal{C}$, and each vertex $v \in \mathcal{C}$ has at most one child w with $\phi_c(v) = \phi_c(w)$ (Equation 3.4), the projection $\phi_c(\text{Min}(\mathcal{H}_r))$ is also equal to γ_r :

$$\phi_c(\mathcal{H}_r) = \phi_c(\Gamma_r) = \phi_c(\text{Min}(\mathcal{H}_r)) = \gamma_r \quad (3.6)$$

3.5.1.3 $\text{Red}()$

Define $\text{Red}(r)$ as the set of all vertices that have red pebbles just before subcomputation \mathcal{H}_r starts, with $\text{Red}(1) = \emptyset$. We further have $\text{Red}(P), P \subset V$ is the set of all vertices in some subset P that have red pebbles and

$Red(\phi_c(P))$ is a set of unique pairs of first two coordinates of vertices in P that have red pebbles.

3.5.1.4 Greedy schedule

We call a schedule $\mathcal{S} = \{\mathcal{H}_1, \dots, \mathcal{H}_h\}$ *greedy* if during every subcomputation \mathcal{H}_r every vertex u that will hold a red pebble either has a child in \mathcal{H}_r or belongs to \mathcal{H}_r :

$$\forall_r : Red(r) \subset \alpha_{r-1} \cup \beta_{r-1} \cup \mathcal{H}_{r-1} \quad (3.7)$$

3.5.2 I/O Optimality of Greedy Schedules

Lemma 3.5. *Any greedy schedule that multiplies matrices of sizes $M \times K$ and $K \times N$ using MNK multiplications requires a minimum number of $\frac{2MNK}{\sqrt{S}} + MN$ I/O operations.*

Proof. We start by creating an X-partition for an MMM CDAG (the values of Y and $R(S)$ are parameters that we determine in the course of the proof). The proof is divided into the following 6 steps (Sections 3.5.2.1 to 3.5.2.6).

3.5.2.1 Red Pebbles During and After Subcomputation

Observe that each vertex in $c = (t_1, t_2, t_3) \in \mathcal{C}, t_1 = 1 \dots M, t_2 = 1 \dots N, t_3 = 1 \dots K - 1$ has only one child $c = (t_1, t_2, t_3 + 1)$. Therefore, we can assume that in an optimal schedule there are no two vertices $(t_1, t_2, t_3), (t_1, t_2, t_3 + f) \in \mathcal{C}, f \in \mathbb{N}_+$ that simultaneously hold a red vertex, as when the vertex $(t_1, t_2, t_3 + 1)$ is pebbled, a red pebble can be immediately removed from (t_1, t_2, t_3) :

$$|Red(\mathcal{H}_r)| = |\phi_c(Red(\mathcal{H}_r))| \quad (3.8)$$

On the other hand, for every vertex v , if all its predecessors $Pred(v)$ have red pebbles, then vertex v may be immediately computed, freeing a red pebble from its predecessor $w \in \mathcal{C}$, due to the fact, that v is the only child of w :

$$\forall_{v \in V} \forall_r : Pred(v) \subset Dom(\mathcal{H}_r) \cup \mathcal{H}_r \implies \exists_{t \leq r} v \in \mathcal{H}_t \quad (3.9)$$

Furthermore, after subcomputation \mathcal{H}_r , all vertices in \mathcal{H}_r that have red pebbles are in its minimum set:

$$Red(r+1) \cap \mathcal{H}_r = Red(r+1) \cap Min(\mathcal{H}_r) \quad (3.10)$$

Combining this result with the definition of a greedy schedule (Equation 3.7), we have

$$\text{Red}(r+1) \subseteq \alpha_r \cup \beta_r \cup \text{Min}(\mathcal{H}_r) \quad (3.11)$$

3.5.2.2 Surface and volume of subcomputations

By the definition of X -partition, the computation is divided into $H(X)$ subcomputations $\mathcal{H}_r \subset \mathcal{C}$, $r \in \{1, \dots, H(X)\}$, such that $\text{Dom}(\mathcal{H}_r), \text{Min}(\mathcal{H}_r) \leq X$.

Inserting Equations 3.5, 3.6, and 3.8, we have:

$$\begin{aligned} |\text{Dom}(\mathcal{H}_r)| &= |\alpha_r| + |\beta_r| + |\gamma_r| \leq X \\ |\text{Min}(\mathcal{H}_r)| &= |\gamma_r| \leq X \end{aligned} \quad (3.12)$$

On the other hand, the Loomis-Whitney inequality [131] bounds the volume of \mathcal{H}_r :

$$|\mathcal{H}_r| \leq \sqrt{|\alpha_r| |\beta_r| |\gamma_r|} \quad (3.13)$$

Consider sets of all different indices accessed by projections $\alpha_r, \beta_r, \gamma_r$:

$$T_1 = \{t_{1,1}, \dots, t_{1,a}\}, |T_1| = a$$

$$T_2 = \{t_{2,1}, \dots, t_{2,b}\}, |T_2| = b$$

$$T_3 = \{t_{3,1}, \dots, t_{3,c}\}, |T_3| = c$$

$$\alpha_r \subseteq \{(t_1, t_3) : t_1 \in T_1, t_3 \in T_3\} \quad (3.14)$$

$$\beta_r \subseteq \{(t_3, t_2) : t_3 \in T_3, t_2 \in T_2\} \quad (3.15)$$

$$\gamma_r \subseteq \{(t_1, t_2) : t_1 \in T_1, t_2 \in T_2\} \quad (3.16)$$

$$\mathcal{H}_r \subseteq \{(t_1, t_2, t_3) : t_1 \in T_1, t_2 \in T_2, t_3 \in T_3\} \quad (3.17)$$

For fixed sizes of the projections $|\alpha_r|, |\beta_r|, |\gamma_r|$, then the volume $|\mathcal{H}_r|$ is maximized when left and right side of Inequalities 3.14 to 3.16 are equal. Using 3.5 and 3.9 we have that 3.17 is an equality too, and:

$$|\alpha_r| = ac, |\beta_r| = bc, |\gamma_r| = ab, |\mathcal{H}_r| = abc, \quad (3.18)$$

achieving the upper bound (Equation 3.13).

3.5.2.3 Reuse set $\mathcal{H}_{R,r}$ and store set $W_{B,r}$

Consider two subsequent computations, \mathcal{H}_r and \mathcal{H}_{r+1} . After \mathcal{H}_r , α_r , β_r , and γ_r may have red pebbles (Equation 3.7). On the other hand, for the dominator set of \mathcal{H}_{r+1} we have $|\text{Dom}(\mathcal{H}_{r+1})| = |\alpha_{r+1}| + |\beta_{r+1}| + |\gamma_{r+1}|$. Then, the reuse set $\mathcal{H}_{R,r+1}$ is an intersection of those sets. Since $\alpha_r \cap \beta_r = \alpha_r \cap \gamma_r = \beta_r \cap \gamma_r = \emptyset$, we have (confront Equation 3.11):

$$\begin{aligned} \mathcal{H}_{R,r+1} &\subseteq (\alpha_r \cap \alpha_{r+1}) \cup (\beta_r \cap \beta_{r+1}) \cup (\text{Min}(\mathcal{H}_r) \cap \Gamma_{r+1}) \\ |\mathcal{H}_{R,r+1}| &\leq |\alpha_r \cap \alpha_{r+1}| + |\beta_r \cap \beta_{r+1}| + |\gamma_r \cap \gamma_{r+1}| \end{aligned} \quad (3.19)$$

Note that vertices in α_r and β_r are inputs of the computation: therefore, by the definition of the red-blue pebble game, they start in the slow memory (they already have blue pebbles). $\text{Min}(\mathcal{H}_r)$, on the other hand, may have only red pebbles placed on them. Furthermore, by the definition of the S -partition, these vertices have children that have not been pebbled yet. They either have to be reused forming the reuse set $\mathcal{H}_{R,r+1}$, or stored back, forming $W_{B,r}$ and requiring the placement of the blue pebbles. Because $\text{Min}(\mathcal{H}_r) \in \mathcal{C}$ and $\mathcal{C} \cap \mathcal{A} = \mathcal{C} \cap \mathcal{B} = \emptyset$, we have:

$$\begin{aligned} W_{B,r} &\subseteq \text{Min}(\mathcal{H}_r) \setminus \Gamma_{r+1} \\ |W_{B,r}| &\leq |\gamma_r \setminus \gamma_{r+1}| \end{aligned} \quad (3.20)$$

3.5.2.4 Overlapping computations

Consider two subcomputations \mathcal{H}_r and \mathcal{H}_{r+1} . Denote shared parts of their projections as $\alpha_s = \alpha_r \cap \alpha_{r+1}$, $\beta_s = \beta_r \cap \beta_{r+1}$, and $\gamma_s = \gamma_r \cap \gamma_{r+1}$. Then, there are two possibilities:

1. \mathcal{H}_r and \mathcal{H}_{r+1} are not cubic, resulting in their volume smaller than the upper bound $|\mathcal{H}_{r+1}| < \sqrt{|\alpha_{r+1}||\beta_{r+1}||\gamma_{r+1}|}$ (Equation 3.13),
2. \mathcal{H}_r and \mathcal{H}_{r+1} are cubic. If all overlapping projections are not empty, then they generate an overlapping computation, that is, there exist vertices v , such that $\phi_{ik}(v) \in \alpha_s$, $\phi_{kj}(v) \in \beta_s$, $\phi_{ij}(v) \in \gamma_s$. Because we consider greedy schedules, those vertices cannot belong to computation \mathcal{H}_{r+1} (Equation 3.9). Therefore, again $|\mathcal{H}_{r+1}| < \sqrt{|\alpha_{r+1}||\beta_{r+1}||\gamma_{r+1}|}$. Now consider sets of all different indices accessed by those rectangular projections (Section 3.5.2.2, Inequalities 3.14 to 3.16). Fixing two non-empty projections we define all

three sets T_1, T_2, T_3 , which in turn, generate the third (non-empty) projection, resulting again in overlapping computations which reduce the size of $|\mathcal{H}_{r+1}|$. Therefore, for cubic subcomputations, their volume is maximized $|\mathcal{H}_{r+1}| = \sqrt{|\alpha_{r+1}||\beta_{r+1}||\gamma_{r+1}|}$ if at most one of the overlapping projections is non-empty (and therefore, there is no overlapping computation).

3.5.2.5 Maximizing computational intensity

Computational intensity ρ_r of a subcomputation \mathcal{H}_r is an upper bound on ratio between its size $|\mathcal{H}_r|$ and the number of I/O operations required. The number of I/O operations is minimized when ρ is maximized (Lemma 3.4):

$$\begin{aligned} \text{maximize } \rho_r &= \frac{|\mathcal{H}_r|}{X - R(S) + T(S)} \geq \frac{|\mathcal{H}_r|}{\text{Dom}(\mathcal{H}_r) - |\mathcal{H}_{R,r}| + |W_{B,r}|} \\ &\text{subject to:} \\ &|\text{Dom}(\mathcal{H}_r)| \leq X \\ &|\mathcal{H}_{R,r}| \leq S \end{aligned}$$

To maximize the computational intensity, for a fixed number of I/O operations, the subcomputation size $|\mathcal{H}_r|$ is maximized. Based on Observation 3.5.2.4, it is maximized only if at most one of the overlapping projections $\alpha_r \cap \alpha_{r+1}, \beta_r \cap \beta_{r+1}, \gamma_r \cap \gamma_{r+1}$ is not empty. Inserting Equations 3.13, 3.12, 3.19, and 3.20, we have the following three equations for the computational intensity, depending on the non-empty projection:

$$\begin{aligned} &\alpha_r \cap \alpha_{r+1} \neq \emptyset : \\ \rho_r &= \frac{\sqrt{|\alpha_r||\beta_r||\gamma_r|}}{|\alpha_r| + |\beta_r| + |\gamma_r| - |\alpha_r \cap \alpha_{r+1}| + |\gamma_r|} \end{aligned} \quad (3.21)$$

$$\begin{aligned} &\beta_r \cap \beta_{r+1} \neq \emptyset : \\ \rho_r &= \frac{\sqrt{|\alpha_r||\beta_r||\gamma_r|}}{|\alpha_r| + |\beta_r| + |\gamma_r| - |\beta_r \cap \beta_{r+1}| + |\gamma_r|} \end{aligned} \quad (3.22)$$

$$\begin{aligned} &\gamma_r \cap \gamma_{r+1} \neq \emptyset : \\ \rho_r &= \frac{\sqrt{|\alpha_r||\beta_r||\gamma_r|}}{|\alpha_r| + |\beta_r| + |\gamma_r| - |\gamma_r \cap \gamma_{r+1}| + |\gamma_r \setminus \gamma_{r+1}|} \end{aligned} \quad (3.23)$$

ρ_r is maximized when $\gamma_r = \gamma_{r+1}, \gamma_r \cap \gamma_{r+1} \neq \emptyset, \gamma_r \setminus \gamma_{r+1} = \emptyset$ (Equation 3.23).

Then, inserting Equations 3.18, we have:

$$\begin{aligned} &\text{maximize } \rho_r = \frac{abc}{ac + cb} \\ &\text{subject to:} \\ &ab + ac + cb \leq X \\ &ab \leq S \\ &a, b, c \in \mathbb{N}_+, \end{aligned}$$

where X is a free variable. Simple optimization technique using Lagrange multipliers yields the result:

$$a = b = \lfloor \sqrt{S} \rfloor, c = 1, \quad (3.24)$$

$$|\alpha_r| = |\beta_r| = \lfloor \sqrt{S} \rfloor, |\gamma_r| = \lfloor \sqrt{S} \rfloor^2,$$

$$|\mathcal{H}_r| = \lfloor \sqrt{S} \rfloor^2, X = \lfloor \sqrt{S} \rfloor^2 + 2\lfloor \sqrt{S} \rfloor$$

$$\rho_r = \frac{\lfloor \sqrt{S} \rfloor}{2} \quad (3.25)$$

From now on, to keep the calculations simpler, we use assume that $\sqrt{S} \in \mathbb{N}_+$.

3.5.2.6 MMM I/O complexity of greedy schedules

By the computational intensity corollary:

$$Q \geq \frac{|V|}{\rho} = \frac{2MNK}{\sqrt{S}}$$

This is the I/O cost of putting a red pebble at least once on every vertex in \mathcal{C} . Note however, that we did not put any blue pebbles on the outputs yet (all vertices in \mathcal{C} had only red pebbles placed on them during the execution). By the definition of the red-blue pebble game, we need to place blue pebbles on MN output vertices, corresponding to the output matrix C , resulting in additional MN I/O operations, yielding final bound

$$Q \geq \frac{2MNK}{\sqrt{S}} + MN$$

□

3.5.2.7 Attainability of the Lower Bound

Restricting the analysis to greedy schedules provides explicit information of a state of memory (sets \mathcal{H}_r , $\mathcal{H}_{R,r}$, $W_{B,r}$), and therefore, to a corresponding CDAG pebbling. In Section 3.5.2.5, it is proven that an optimal greedy schedule is composed of $\frac{MNK}{R(S)}$ outer product calculations, while loading $\sqrt{R(S)}$ elements of each of matrices A and B . While the lower bound is achieved for $R(S) = S$, such a schedule is infeasible, as at least some additional red pebbles, except the ones placed on the reuse set $\mathcal{H}_{R,r}$, have to be placed on $2\sqrt{R(S)}$ vertices of A and B .

A direct way to obtain a feasible greedy schedule is to set $X = S$, ensuring that the dominator set can fit into the memory. Then each subcomputation is an outer-product of column-vector of matrix A and row-vector of B , both holding $\sqrt{S+1} - 1$ values. Such a schedule performs $\frac{2MNK}{\sqrt{S+1}-1} + MN$ I/O operations, a factor of $\frac{\sqrt{S}}{\sqrt{S+1}-1}$ more than a lower bound, which quickly approach 1 for large S . Listing 3.1 provides a pseudocode of this algorithm, which is a well-known rank-1 update formulation of MMM. However, we can do better.

Let's consider a generalized case of such subcomputation \mathcal{H}_r . Assume, that in each step:

1. a elements of A (forming α_r) are loaded,
2. b elements of B (forming β_r) are loaded,
3. ab partial results of C are kept in the fast memory (forming Γ_r)
4. ab values of C are updated (forming \mathcal{H}_r),
5. no store operations are performed.

Each vertex in α_r has b children in \mathcal{H}_r (each of which has also a parent in β_r). Similarly, each vertex in β_r has a children in \mathcal{H}_r , each of which has also a parent in α_r . We first note, that $ab < S$ (otherwise, we cannot do any computation while keeping all ab partial results in fast memory). Any red vertex placed on α_r should not be removed from it until all b children are pebbled, requiring red-pebbling of corresponding b vertices from β_r . But, in turn, any red pebble placed on a vertex in β_r should not be removed until all a children are red pebbled.

Therefore, either all a vertices in α_r , or all b vertices in β_r have to be hold red pebbles at the same time, while at least one additional red pebble is

needed on β_r (or α_r). W.l.o.g., assume we keep red pebbles on all vertices of α_r . We then have:

$$\begin{aligned} & \text{maximize } \rho_r = \frac{ab}{a+b} \\ & \text{subject to:} \\ & ab + a + 1 \leq S \\ & a, b \in \mathbb{N}_+, \end{aligned} \tag{3.26}$$

The solution to this problem is

$$a_{opt} = \left\lfloor \frac{\sqrt{(S-1)^3 - S + 1}}{S-2} \right\rfloor < \sqrt{S} \tag{3.27}$$

$$b_{opt} = \left\lfloor -\frac{2S + \sqrt{(S-1)^3 - S^2 - 1}}{\sqrt{(S-1)^3 - S + 1}} \right\rfloor < \sqrt{S} \tag{3.28}$$

Listing 3.1: Pseudocode of near optimal sequential MMM

```

1  for  $i_1 = 1 : \left\lceil \frac{M}{a_{opt}} \right\rceil$ 
2  for  $j_1 = 1 : \left\lceil \frac{N}{b_{opt}} \right\rceil$ 
3    for  $r = 1 : K$ 
4      for  $i_2 = i_1 \cdot T : \min((i_1 + 1) \cdot a_{opt}, M)$ 
5        for  $j_2 = j_1 \cdot T : \min((j_1 + 1) \cdot b_{opt}, N)$ 
6           $C(i_2, j_2) = C(i_2, j_2) + A(i_2, r) \cdot B(r, j_2)$ 

```

3.5.3 Greedy vs Non-greedy Schedules

In Section 3.5.2.6, it is shown that the I/O lower bound for any greedy schedule is $Q \geq \frac{2MNK}{\sqrt{S}} + MN$. Furthermore, Listing 3.1 provide a schedule that attains this lower bound (up to a $a_{opt}b_{opt}/S$ factor). To prove that this bound applies to any schedule, we need to show, that any non-greedy can not perform better (perform less I/O operations) than the greedy schedule lower bound.

Lemma 3.6. *Any non-greedy schedule computing classical matrix multiplication performs at least $\frac{2MNK}{\sqrt{S}} + MN$ I/O operations.*

Proof. Lemma 3.3 applies to any schedule and for any value of X . Clearly, for any general schedule we cannot directly model $\mathcal{H}_{R,i}$, $\mathcal{H}_{B,i}$, $W_{R,i}$, and $W_{B,i}$, and therefore $T(S)$ and $R(S)$. However, it is always true that $0 \leq T(S)$ and $R(S) \leq S$. Also, the dominator set formed in Equation 3.5 applies for any subcomputation, as well as a bound on $|\mathcal{H}_r|$ from Inequality 3.13. We can then rewrite the computational intensity maximization problem:

$$\begin{aligned} \text{maximize } \rho_r &= \frac{|\mathcal{H}_r|}{X - R(S) + T(S)} \leq \frac{\sqrt{|\alpha_r||\beta_r||\gamma_r|}}{|\alpha_r| + |\beta_r| + |\gamma_r| - S} \\ &\text{subject to:} \\ S &< |\alpha_r| + |\beta_r| + |\gamma_r| = X \end{aligned} \tag{3.29}$$

This is maximized for $|\alpha_r| = |\beta_r| = |\gamma_r| = X/3$, yielding

$$\rho_r = \frac{(X/3)^{3/2}}{X - S}$$

Because MNK/ρ_r is a valid lower bound for any $X > S$ (Lemma 3.4), we want to find such value X_{opt} for which ρ_r is minimal, yielding the highest (tightest) lower bound on Q :

$$\begin{aligned} \text{minimize } \rho_r &= \frac{(X/3)^{3/2}}{X - S} \\ &\text{subject to:} \\ X &\geq S \end{aligned} \tag{3.30}$$

which, in turn, is minimized for $X = 3S$. This again shows, that the upper bound on maximum computational intensity for any schedule is $\sqrt{S}/2$, which matches the bound for greedy schedules (Equation 3.25). \square

We note that Smith and van de Gein [56] in their paper also bounded the number of computations (interpreted geometrically as a subset in a 3D space) by its surface and obtained an analogous result for this surface (here, a dominator and minimum set sizes). However, using computational intensity lemma, our bound is tighter by $2S$ ($+MN$, counting storing the final result).

Proof of Theorem 3.1:

Lemma 3.5 establishes that the I/O lower bound for any greedy schedule is $Q = 2MNK/\sqrt{S} + MN$. Lemma 3.6 establishes that no other schedule can perform less I/O operations. \square

Corollary: The greedy schedule associated with an $X = S$ -partition performs at most $\frac{\sqrt{S}}{\sqrt{S+1}-1}$ more I/O operations than a lower bound. The optimal greedy schedule is associated with an $X = a_{opt}b_{opt} + a_{opt} + b_{opt}$ -partition and performs $\frac{\sqrt{S}(a_{opt}+b_{opt})}{a_{opt}b_{opt}}$ I/O operations.

3.6 OPTIMAL PARALLEL MMM

We now derive the schedule of COSMA from the results from Section 3.5.2.7. The key notion is the data reuse, that determines not only the sequential execution, as discussed in Section 3.4.2, but also the parallel scheduling. Specifically, if the data reuse set spans across multiple local domains, then this set has to be communicated between these domains, increasing the I/O cost (Figure 3.3). We first introduce a formalism required to parallelize the sequential schedule (Section 3.6.1). In Section 3.6.2, we generalize parallelization strategies used by the 2D, 2.5D, and recursive decompositions, deriving their communication cost and showing that none of them is optimal in the whole range of parameters. We finally derive the optimal decomposition (*FindOptimalDomain* function in Algorithm 1) by expressing it as an optimization problem (Section 3.6.3), and analyzing its I/O and latency cost. The remaining steps in Algorithm 1: *FitRanks*, *GetDataDecomp*, as well as *DistrData* and *Reduce* are discussed in Section 3.7.1, Section 3.7.6, and Section 3.7.2, respectively. For a distributed machine, we assume that all matrices fit into collective memories of all processors: $pS \geq MN + MK + NK$. For a shared memory setting, we assume that all inputs start in a common slow memory.

3.6.1 Sequential and Parallel Schedules

We now describe how a parallel schedule is formed from a sequential one. The sequential schedule \mathcal{S} partitions the CDAG $G = (V, E)$ into $H(\mathcal{S})$ subcomputations \mathcal{H}_i . The parallel schedule \mathcal{P} divides \mathcal{S} among p processors: $\mathcal{P} = \{\mathcal{D}_1, \dots, \mathcal{D}_p\}, \bigcup_{j=1}^p \mathcal{D}_j = \mathcal{S}$. The set \mathcal{D}_j of all \mathcal{H}_k assigned to processor j forms a *local domain* of j (Fig. 3.4c).

If two local domains \mathcal{D}_k and \mathcal{D}_l are dependent, that is, $\exists u, \exists v : u \in \mathcal{D}_k \wedge v \in \mathcal{D}_l \wedge (u, v) \in E$, then u has to be *communicated* from processor K to l . The total number of vertices communicated between all processors is the *I/O cost* Q of schedule \mathcal{P} . We say that the parallel schedule \mathcal{P}_{opt} is *communication-optimal* if $Q(\mathcal{P}_{opt})$ is minimal among all possible parallel schedules.

The vertices of MMM CDAG may be arranged in an $[M \times N \times K]$ 3D grid called an *iteration space* [48]. The orthonormal vectors $\mathbf{i}, \mathbf{j}, \mathbf{K}$ correspond to the loops in Lines 1-3 in Listing 3.1 (Figure 3a). We call a schedule \mathcal{P} *parallelized in dimension \mathbf{d}* if we “cut” the CDAG along dimension \mathbf{d} . More formally, each local domain $\mathcal{D}_j, j = 1 \dots p$ is a grid of size either $[M/p, N, K]$, $[M, N/p, K]$, or $[M, N, K/p]$. The schedule may also be parallelized in two dimensions ($\mathbf{d}_1 \mathbf{d}_2$) or three dimensions ($\mathbf{d}_1 \mathbf{d}_2 \mathbf{d}_3$) with a local domain size $[M/p_m, N/p_n, K/p_k]$ for some p_m, p_n, p_k , such that $p_m p_n p_k = p$. We call $\mathcal{G} = [p_m, p_n, p_k]$ the *processor grid* of a schedule. E.g., Cannon’s algorithm is parallelized in dimensions \mathbf{ij} , with the processor grid $[\sqrt{p}, \sqrt{p}, 1]$. COSMA, on the other hand, may use any of the possible parallelizations, depending on the problem parameters.

3.6.2 Parallelization Strategies for MMM

The sequential schedule \mathcal{S} (Section 3.5) consists of MNK/S elementary outer product calculations, arranged in $\sqrt{S} \times \sqrt{S} \times K$ “blocks” (Figure 3.4). The number $p_1 = MN/S$ of dependency-free subcomputations \mathcal{H}_i (i.e., having no parents except for input vertices) in \mathcal{S} determines the maximum degree of parallelism of \mathcal{P}_{opt} for which no reuse set $\mathcal{H}_{R,i}$ crosses two local domains $\mathcal{D}_j, \mathcal{D}_k$. The optimal schedule is parallelized in dimensions \mathbf{ij} . There is no communication between the domains (except for inputs and outputs), and all I/O operations are performed inside each \mathcal{D}_j following the sequential schedule. Each processor is assigned to p_1/p local domains \mathcal{D}_j of size $[\sqrt{S}, \sqrt{S}, K]$, each of which requires $2\sqrt{S}K + S$ I/O operations (Theorem 3.1), giving a total of $Q = 2MNK/(p\sqrt{S}) + MN/p$ I/O operations per processor.

When $p > p_1$, the size of local domains $|\mathcal{D}_j|$ is smaller than $\sqrt{S} \times \sqrt{S} \times K$. Then, the schedule has to either be parallelized in dimension \mathbf{K} , or has to reduce the size of the domain in \mathbf{ij} plane. The former option creates dependencies between the local domains, which results in additional communication (Figure 3.4e). The latter does not utilize the whole available memory, making the sequential schedule not I/O optimal and decreasing

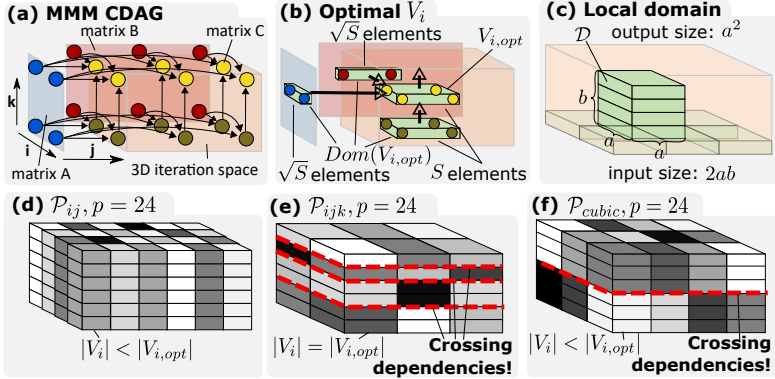


Figure 3.4: (a) An MMM CDAG as a 3D grid (iteration space). Each vertex in it (except for the vertices in the bottom layer) has three parents - blue (matrix A), red (matrix B), and yellow (partial result of matrix C) and one yellow child (except for vertices in the top layer). (b) A union of inputs of all vertices in \mathcal{H}_i form the dominator set $Dom(\mathcal{H}_i)$ (two blue, two red and four dark yellow). Using approximation $\sqrt{S+1} - 1 \approx \sqrt{S}$, we have $|Dom(\mathcal{H}_{i,opt})| = S$. (c) A local domain \mathcal{D} consists of b subcomputations \mathcal{H}_i , each of a dominator size $|Dom(\mathcal{H}_i)| = a^2 + 2a$. (d-f) Different parallelization schemes of near optimal sequential MMM for $p = 24 > p_1 = 6$.

the computational intensity ρ (Figure 3.4d). We now analyze three possible parallelization strategies (Figure 3.4) which generalize 2D, 2.5D, and recursive decomposition strategies; see Table 3.3 for details.

Schedule \mathcal{P}_{ij} . The schedule is parallelized in dimensions i and j . The processor grid is $\mathcal{G}_{ij} = [\frac{M}{a}, \frac{N}{a}, 1]$, where $a = \sqrt{\frac{MN}{p}}$. Because all dependencies are parallel to dimension K , there are no dependencies between \mathcal{D}_j except for the inputs and the outputs. Because $a < \sqrt{S}$, the corresponding sequential schedule has a reduced computational intensity $\rho_{ij} < \sqrt{S}/2$.

Schedule \mathcal{P}_{ijk} . The schedule is parallelized in all dimensions. The processor grid is $\mathcal{G}_{ijk} = [\frac{M}{\sqrt{S}}, \frac{N}{\sqrt{S}}, \frac{pS}{MN}]$. The computational intensity $\rho_{ijk} = \sqrt{S}/2$ is optimal. The parallelization in K dimension creates dependencies between local domains, requiring communication and increasing the I/O cost.

Schedule \mathcal{P}_{cubic} . The schedule is parallelized in all dimensions. The grid is $[\frac{M}{a_c}, \frac{N}{a_c}, \frac{K}{a_c}]$, where $a_c = \min \left\{ \left(\frac{MNK}{p} \right)^{1/3}, \sqrt{\frac{S}{3}} \right\}$. Because $a_c < \sqrt{S}$, the corresponding computational intensity $\rho_{cubic} < \sqrt{S}/2$ is not optimal. The

parallelization in \mathbf{K} dimension creates dependencies between local domains, increasing communication.

Schedules of the State-of-the-Art Decompositions. If $M = N$, the \mathcal{P}_{ij} scheme is reduced to the classical 2D decomposition (e.g., Cannon's algorithm [60]), and \mathcal{P}_{ijk} is reduced to the 2.5D decomposition [52]. CARMA [64] asymptotically reaches the \mathcal{P}_{cubic} scheme, guaranteeing that the longest dimension of a local cuboidal domain is at most two times larger than the smallest one. We present a detailed complexity analysis comparison for all algorithms in Table 3.3.

3.6.3 I/O Optimal Parallel Schedule

Observe that none of those schedules is optimal in the whole range of parameters. As discussed in Section 3.5, in sequential scheduling, intermediate results of C are not stored to the memory: they are consumed (reused) immediately by the next sequential step. Only the final result of C in the local domain is sent. Therefore, the optimal parallel schedule \mathcal{P}_{opt} minimizes the communication, that is, sum of the inputs' sizes plus the output size, under the sequential I/O constraint on subcomputations $\forall \mathcal{H}_i \in \mathcal{D}_j \in \mathcal{P}_{opt} |Dom(\mathcal{H}_i)| \leq S \wedge |Min(\mathcal{H}_i)| \leq S$.

The local domain \mathcal{D}_j is a grid of size $[a \times a \times b]$, containing b outer products of vectors of length a . The optimization problem of finding \mathcal{P}_{opt} using the computational intensity (Lemma 3.4) is formulated as follows:

$$\begin{aligned} & \text{maximize } \rho = \frac{a^2 b}{ab + ab + a^2} & (3.31) \\ & \text{subject to:} \\ & a^2 \leq S \text{ (the I/O constraint)} \\ & a^2 b = \frac{MNK}{p} \text{ (the load balance constraint)} \\ & pS \geq MN + MK + NK \text{ (matrices must fit into memory)} \end{aligned}$$

The I/O constraint $a^2 \leq S$ is binding (changes to equality) for $p \leq \frac{MNK}{S^{3/2}}$. Therefore, the solution to this problem is:

$$a = \min \left\{ \sqrt{S}, \left(\frac{MNK}{p} \right)^{1/3} \right\}, \quad b = \max \left\{ \frac{MNK}{pS}, \left(\frac{MNK}{p} \right)^{1/3} \right\} \quad (3.32)$$

Decomposition	2D [62]	2.5D [52]	recursive [64]	COSMA (this work)
Parallel schedule \mathcal{P}	\mathcal{P}_{ij} for $M = N$	\mathcal{P}_{ijk} for $M = N$	\mathcal{P}_{cubic}	\mathcal{P}_{opt}
grid $[p_n \times p_n \times p_n]$	$[\sqrt{p} \times \sqrt{p} \times 1]$	$[\sqrt{p/c} \times \sqrt{p/c} \times c]; c = \frac{p^S}{MK+NR}$	$[2^{n_1} \times 2^{n_2} \times 2^{n_3}];$ $a_1 + a_2 + a_3 = \log_2(p)$	$[\frac{M}{a} \times \frac{N}{b} \times \frac{K}{c}];$ a, b, c : Equation 3.32
domain size	$[\frac{M}{\sqrt{p}} \times \frac{N}{\sqrt{p}} \times K]$	$[\frac{M}{\sqrt{p/c}} \times \frac{N}{\sqrt{p/c}} \times \frac{K}{c}]$	$[\frac{M}{2^{n_1}} \times \frac{N}{2^{n_2}} \times \frac{K}{2^{n_3}}]$	$[a \times a \times b]$
I/O cost Q	$\frac{K}{\sqrt{p}}(M+N) + \frac{MN}{p}$	$\frac{(K(M+N))^{3/2}}{p\sqrt{S}} + \frac{MNS}{K(M+N)}$	$Q \geq \begin{cases} 2\sqrt{3} \frac{MNK}{p\sqrt{S}} + \left(\frac{MNK}{p}\right)^{2/3} & p \leq \frac{3\sqrt{3}MNK}{S^{3/2}} \\ 3\left(\frac{MNK}{p}\right)^{2/3} & p > \frac{3\sqrt{3}MNK}{S^{3/2}} \end{cases}$	$Q \geq \begin{cases} \frac{2MNK}{p\sqrt{S}} + S & p \leq \frac{MNK}{S^{3/2}} \\ 3\left(\frac{MNK}{p}\right)^{2/3} & p > \frac{MNK}{S^{3/2}} \end{cases}$
latency cost L	$2k \log_2(\sqrt{p})$	$\frac{(K(M+N))^{5/2}}{pS^{3/2}(kn+kr-MN)} + 3 \log_2\left(\frac{p^S}{MK+NR}\right)$	$(3^{3/2}MNK) / (pS^{3/2}) + 3 \log_2(p)$	$\frac{2nb}{S-p^2} \log_2\left(\frac{MN}{p^2}\right)$
I/O cost Q	$2n^2(\sqrt{p}+1)/p$	$2n^2(\sqrt{p}+1)/p$	$2n^2(\sqrt{3/2p}+1/2p^{2/3})$	$2n^2(\sqrt{p}+1)/p$
latency cost L	$2k \log_2(\sqrt{p})$	\sqrt{p}	$(\frac{3}{2})^{3/2} \sqrt{p} \log_2(p)$	$\sqrt{p} \log_2(p)$
I/O cost	$p^{3/2}/2$	$p^{4/3}/2 + p^{1/3}$	$3p/4$	$p(3-2^{1/3})/2^{4/3} \approx 0.69p$
latency cost L	$p^{3/2} \log_2(\sqrt{p})/4$	1	1	1

"General case":

Square matrices, "limited memory": $M = N = K, S = 2N^2/p, p = 2^{3n}$

"Tall" matrices, "extra" memory available: $M = N = \sqrt{p}, K = p^{3/2}/4, S = 2NK/p^{2/3}, p = 2^{3n+1}$

Table 3.3: The comparison of complexities of 2D, 2.5D, recursive, and COSMA algorithms. The 3D decomposition is a special case of 2.5D, and can be obtained by instantiating $c = p^{1/3}$ in the 2.5D case. In addition to the general analysis, we show two special cases. If the matrices are square and there is no extra memory available, 2D, 2.5D and COSMA achieves tight communication lower bound $2n^2/\sqrt{p}$, whereas CARMA performs $\sqrt{3}$ times more communication. If one dimension is much larger than the others and there is extra memory available, 2D, 2.5D and CARMA decompositions perform $\mathcal{O}(p^{1/2})$, $\mathcal{O}(p^{1/3})$, and 8% more communication than COSMA, respectively. For simplicity, we assume that parameters are chosen such that all divisions have integer results.

The I/O complexity of this schedule is:

$$Q \geq \frac{a^2 b}{\rho} = \begin{cases} \frac{2MNK}{p\sqrt{S}} + S & p \leq \frac{MNK}{S^{3/2}} \\ 3\left(\frac{MNK}{p}\right)^{2/3} & p > \frac{MNK}{S^{3/2}} \end{cases} \quad (3.33)$$

This can be intuitively interpreted geometrically as follows: if we imagine the optimal local domain "growing" with the decreasing number of processors, then it stays cubic as long as it is still "small enough" (its side is smaller than \sqrt{S}). After that point, its face in the ij plane stays constant $\sqrt{S} \times \sqrt{S}$ and it "grows" only in the K dimension. This schedule effectively switches from \mathcal{P}_{ijk} to \mathcal{P}_{cubic} once there is enough memory ($S \geq (MNK/p)^{2/3}$).

Theorem 3.2. *The I/O complexity of a classic matrix multiplication algorithm executed on p processors, each of local memory size $S \geq \frac{MN+MK+NK}{p}$ is*

$$Q \geq \begin{cases} \frac{2MNK}{p\sqrt{S}} + S & p \leq \frac{MNK}{S^{3/2}} \\ 3\left(\frac{MNK}{p}\right)^{2/3} & p > \frac{MNK}{S^{3/2}} \end{cases}$$

Proof. The theorem is a direct consequence of Lemma 3.3 and the computational intensity (Lemma 3.4). The load balance constraint enforces a size of each local domain $|\mathcal{D}_j| = MNK/p$. The I/O cost is then bounded by $|\mathcal{D}_j|/\rho$. Schedule \mathcal{P}_{opt} maximizes ρ by the formulation of the optimization problem (Equation 3.31). \square

I/O-Latency Trade-off. As showed in this section, the local domain \mathcal{D} of the near optimal schedule \mathcal{P} is a grid of size $[a \times a \times b]$, where a, b are given by Equation (3.32). The corresponding sequential schedule \mathcal{S} is a sequence of b outer products of vectors of length a . Denote the size of the communicated inputs in each step by $I_{step} = 2a$. This corresponds to b steps of communication (the latency cost is $L = b$).

The number of steps (latency) is equal to the total communication volume of \mathcal{D} divided by the volume per step $L = Q/I_{step}$. To reduce the latency, one either has to decrease Q or increase I_{step} , under the memory constraint that $I_{step} + a^2 \leq S$ (otherwise we cannot fit both the inputs and the outputs in the memory). Express $I_{step} = a \cdot h$, where h is the number of sequential

subcomputations \mathcal{H}_i we merge in one communication. We can express the I/O-latency trade-off:

$$\begin{aligned} & \min(Q, L) \\ & \text{subject to:} \\ & Q = 2ab + a^2, L = \frac{b}{h} \\ & a^2 + 2ah \leq S \text{ (I/O constraint)} \\ & a^2b = \frac{MNK}{p} \text{ (load balance constraint)} \end{aligned}$$

Solving this problem, we have $Q = \frac{2MNK}{pa} + a^2$ and $L = \frac{2MNK}{pa(S-a^2)}$, where $a \leq \sqrt{S}$. Increasing a we *reduce* the I/O cost Q and *increase* the latency cost L . For minimal value of Q (Theorem 3.2), $L = \left\lceil \frac{2ab}{S-a^2} \right\rceil$, where $a = \min\{\sqrt{S}, (MNK/p)^{1/3}\}$ and $b = \max\{\frac{MNK}{pS}, (MNK/p)^{1/3}\}$. Based on our experiments, we observe that the I/O cost is vastly greater than the latency cost, therefore our schedule by default minimizes Q and uses extra memory (if any) to reduce L .

3.7 IMPLEMENTATION

We now present implementation optimizations that further increase the performance of COSMA on top of the speedup due to our near I/O optimal schedule. The algorithm is designed to facilitate the overlap of communication and computation Section 3.7.3. For this, to leverage the RDMA mechanisms of current high-speed network interfaces, we use the MPI one-sided interface Section 3.7.4. In addition, our implementation also offers alternative efficient two-sided communication back end that uses MPI collectives. We also use a blocked data layout Section 3.7.6, a grid-fitting technique Section 3.7.1, and an optimized binary broadcast tree using static information about the communication pattern (Section 3.7.2) together with the buffer swapping (Section 3.7.5). For the local matrix operations, we use BLAS routines for highest performance. Our code is publicly available at <https://github.com/eth-cscs/COSMA>.

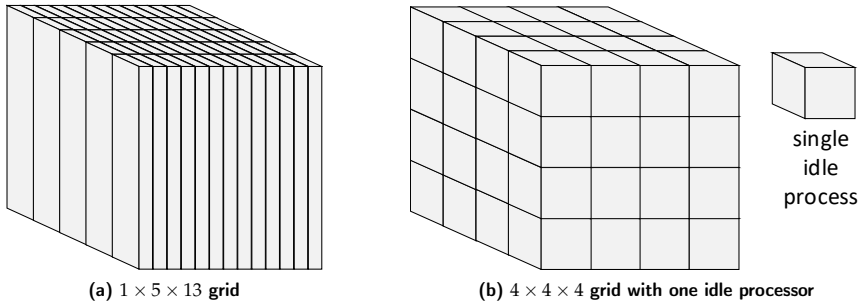


Figure 3.5: Processor decomposition for square matrices and 65 processors. (a) To utilize all resources, the local domain is drastically stretched. (b) Dropping one processor results in a symmetric grid which increases the computation per processor by 1.5%, but reduces the communication by 36%.

3.7.1 Processor Grid Optimization

Throughout this analysis, we assume all operations required to assess the decomposition (divisions, roots) result in natural numbers. We note that in practice it is rarely the case, as the parameters usually emerge from external constraints, like a specification of a performed calculation or hardware resources (Section 3.8). If matrix dimensions are not divisible by the local domain sizes a, b (Equation 3.32), then a straightforward option is to use the floor function, not utilizing the “boundary” processors whose local domains do not fit entirely in the iteration space, which result in more computation per processor. The other option is to find factors of p and then construct the processor grid by matching the largest factors with largest matrix dimensions. However, if the factors of p do not match M, N , and K , this may result in a suboptimal decomposition. Our algorithm allows to not utilize some processors (increasing the computation volume per processor) to optimize the grid, which reduces the communication volume. Figure 3.5 illustrates the comparison between these options. We balance this communication–computation trade-off by “stretching” the local domain size derived in Section 3.6.3 to fit the global domain by adjusting its width, height, and length. The range of this tuning (how many processors we drop to reduce communication) depends on the hardware specification of the machine (peak flop/s, memory and network bandwidth). For our experiments on the Piz Daint machine we chose the

maximal number of unutilized cores to be 3%, accounting for up to 2.4 times speedup for the square matrices using 2,198 cores (Section 3.9).

3.7.2 *Enhanced Communication Pattern*

As shown in Algorithm 1, COSMA by default executes in $t = \frac{2ab}{S-a^2}$ rounds. In each round, each processor receives $s = ab/t = (S - a^2)/2$ elements of A and B . Thus, the input matrices are broadcast among the i and j dimensions of the processor grid. After the last round, the partial results of C are reduced among the K dimension. The communication pattern is therefore similar to ScaLAPACK or CTF.

To accelerate the collective communication, we implement our own binary broadcast tree, taking advantage of the known data layout, processor grid, and communication pattern. Knowing the initial data layout (Section 3.7.6) and the processor grid (Section 3.7.1), we craft the binary reduction tree in all three dimensions i , j , and K such that the distance in the grid between communicating processors is minimized. Our implementation outperforms the standard MPI broadcast from the Cray-MPICH 3.1 library by approximately 10%.

3.7.3 *Communication–Computation Overlap*

The sequential rounds of the algorithm $t_i = 1, \dots, t$, naturally express communication–computation overlap. Using double buffering, at each round t_i we issue an asynchronous communication (using either MPI_Get or MPI_Isend / MPI_Irecv, Section 3.7.4) of the data required at round t_{i+1} , while locally processing the data received in a previous round. We note that, by the construction of the local domains \mathcal{D}_j (Section 3.6.3), the extra memory required for double buffering is rarely an issue. If we are constrained by the available memory, then the space required to hold the partial results of C , which is a^2 , is much larger than the size of the receive buffers $s = (S - a^2)/2$. If not, then there is extra memory available for the buffering.

Number of rounds: The minimum number of rounds, and therefore latency, is $t = \frac{2ab}{S-a^2}$ (Section 3.6.3). However, to exploit more overlap, we can increase the number of rounds $t_2 > t$. In this way, in one round we communicate less data $s_2 = ab/t_2 < s$, allowing the first round of computation to start earlier.

3.7.4 One-Sided vs Two-Sided Communication

To reduce the latency [132] we implemented communication using MPI RMA [133]. This interface utilizes the underlying features of Remote Direct Memory Access (RDMA) mechanism, bypassing the OS on the sender side and providing zero-copy communication: data sent is not buffered in a temporary address, instead, it is written directly to its location.

All communication windows are pre-allocated using `MPI_Win_allocate` with the size of maximum message in the broadcast tree $2^{s-1}D$ (Section 3.7.2). Communication in each step is performed using the `MPI_Get` and `MPI_Accumulate` routines.

For compatibility reasons, as well as for the performance comparison, we also implemented a communication back-end using MPI two-sided (the message passing abstraction).

3.7.5 Communication Buffer Optimization

The binary broadcast tree pattern is a generalization of the recursive structure of CARMA. However, CARMA in each recursive step dynamically allocates new buffers of the increasing size to match the message sizes $2^{s-1}D$, causing an additional runtime overhead.

To alleviate this problem, we pre-allocate initial, send, and receive buffers for each of matrices A , B , and C of the maximum size of the message ab/t , where $t = \frac{2ab}{s-a^2}$ is the number of steps in COSMA (Algorithm 1). Then, in each level s of the communication tree, we move the pointer in the receive buffer by $2^{s-1}D$ elements.

3.7.6 Blocked Data Layout

COSMA's schedule induces the optimal initial data layout, since for each D_j it determines its dominator set $Dom(D_j)$, that is, elements accessed by processor j . Denote $A_{l,j}$ and $B_{l,j}$ subsets of elements of matrices A and B that initially reside in the local memory of processor j . The optimal data layout therefore requires that $A_{l,j}, B_{l,j} \subset Dom(D_j)$. However, the schedule does not specify exactly which elements of $Dom(D_j)$ should be in $A_{l,j}$ and $B_{l,j}$. As a consequence of the communication pattern (Section 3.7.2), each element of $A_{l,j}$ and $B_{l,j}$ is communicated to g_m, g_n processors, respectively. To prevent data reshuffling, we therefore split each of $Dom(D_j)$ into g_m and g_n smaller blocks, enforcing that consecutive blocks are assigned to

processors that communicate first. This is unlike the distributed CARMA implementation [64], which uses the cyclic distribution among processors in the recursion base case and requires local data reshuffling after each communication round. Another advantage of our blocked data layout is a full compatibility with the block-cyclic one, which is used in other linear-algebra libraries.

3.8 EVALUATION

We evaluate COSMA’s communication volume and performance against other state-of-the-art implementations with various combinations of matrix dimensions and memory requirements. These scenarios include both synthetic square matrices, in which all algorithms achieve their peak performance, as well as “flat” (two large dimensions) and real-world “tall-and-skinny” (one large dimension) cases with uneven number of processors.

Comparison Targets

As a comparison, we use the widely used ScaLAPACK library as provided by Intel MKL (version: 18.0.2.199)², as well as Cyclops Tensor Framework³, and the original CARMA implementation⁴. *We manually tune ScaLAPACK parameters to achieve its maximum performance.* Our experiments showed that on Piz Daint it achieves the highest performance when run with 4 MPI ranks per compute node, 9 cores per rank. Therefore, for each matrix sizes/node count configuration, we recompute the optimal rank decomposition for ScaLAPACK. Remaining implementations use default decomposition strategy and perform best utilizing 36 ranks per node, 1 core per rank.

Infrastructure and Implementation Details

All implementations were compiled using the GCC 6.2.0 compiler. We use Cray-MPICH 3.1 implementation of MPI. The parallelism within a rank of ScaLAPACK⁵ is handled internally by the MKL BLAS (with GNU OpenMP threading) version 2017.4.196. To profile MPI communication volume, we use the mpiP version 3.4.1 [134].

² the latest version available on Piz Daint when benchmarks were performed (August 2018). No improvements of P[S,D,C,Z]GEMM have been reported in the MKL release notes since then.

³ <https://github.com/cyclops-community/ctf>, commit ID 244561c on May 15, 2018

⁴ <https://github.com/lipshitz/CAPS>, commit ID 7589212 on July 19, 2013

⁵ only ScaLAPACK uses multiple cores per ranks

Experimental Setup and Architectures

We run our experiments on the CPU partition of CSCS Piz Daint, which has 1,813 XC40 nodes with dual-socket Intel Xeon E5-2695 v4 processors (2 · 18 cores, 3.30 GHz, 45 MiB L3 shared cache, 64 GiB DDR3 RAM), interconnected by the Cray Aries network with a dragonfly network topology. We set p to a number of available cores⁶ and S to the main memory size per core (Section 3.2.1). To additionally capture cache size per core, the model can be extended to a three-level memory hierarchy. However, cache-size tiling is already handled internally by the MKL.

Matrix Dimensions and Number of Cores

We use square ($M = N = K$), “largeK” ($M = N \ll K$), “largeM” ($M \gg N = K$), and “flat” ($M = N \gg K$) matrices. The matrix dimensions and number of cores are (1) powers of two $M = 2^{r_1}$, $N = 2^{r_2}$, $M = 2^{r_3}$, (2) determined by the real-life simulations or hardware architecture (available nodes on a computer), (3) chosen adversarially, e.g. $N^3 + 1$. Tall matrix dimensions are taken from an application benchmark, namely the calculation of the random phase approximation (RPA) energy of water molecules [40]. There, to simulate w molecules, the sizes of the matrices are $M = N = 136w$ and $K = 228w^2$. In the strong scaling scenario, we use $w = 128$ as in the original paper, yielding $M = N = 17,408$, $K = 3,735,552$. For performance runs, we scale up to 512 nodes (18,432 cores).

Selection of Benchmarks

We perform both strong scaling and *memory scaling* experiments. The memory scaling scenario fixes the input size per core ($\frac{pS}{I}$, $I = MN + MK + NK$), as opposed to the work per core ($\frac{MNK}{p} \neq const$). We evaluate two cases: (1) “limited memory” ($\frac{pS}{I} = const$), and (2) “extra memory” ($\frac{p^{2/3}S}{I} = const$).

To provide more information about the impact of communication optimizations on the total runtime, for each of the matrix shapes we also separately measure time spent by COSMA on different parts of the code. For each matrix shape we present two extreme cases of strong scaling - with smallest number of processors (most compute-intense) and with the largest (most communication-intense). To additionally increase information provided, we perform these measurements with and without communication-computation overlap.

⁶ for ScaLAPACK, actual number of MPI ranks is $p/9$

Programming Models

We use either the RMA or the Message Passing models. CTF also uses both models, whereas CARMA and ScaLAPACK use MPI two-sided (Message Passing).

Experimentation Methodology

For each combination of parameters, we perform 5 runs, each with different node allocation. As all the algorithms use BLAS routines for local matrix computations, for each run we execute the kernels three times and take the minimum to compensate for the BLAS setup overhead. We report median and 95% confidence intervals of the runtimes.

3.9 RESULTS

We now present the experimental results comparing COSMA with the existing algorithms. For both strong and memory scaling, we measure total communication volume and runtime on both square and tall matrices. Our experiments show that COSMA always communicates least data and is the fastest in *all* scenarios.





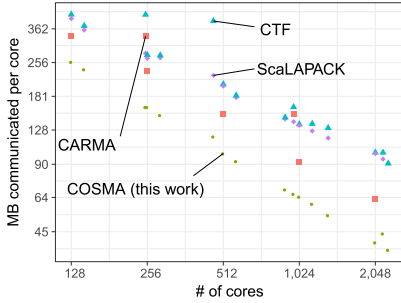
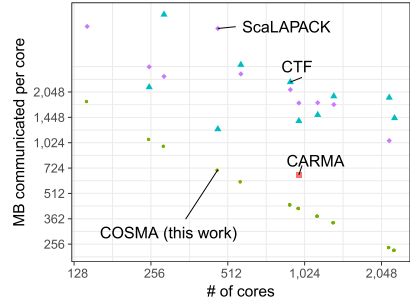
shape	benchmark	total comm. volume per rank [MB]				speedup		
		ScaLAPACK	CTF	CARMA	COSMA	min	mean	max
	strong scaling	203	222	195	107	1.07	1.94	4.81
	limited memory	816	986	799	424	1.23	1.71	2.99
	extra memory	303	350	291	151	1.14	2.03	4.73
	strong scaling	2636	2278	659	545	1.24	2.00	6.55
	limited memory	368	541	128	88	1.30	2.61	8.26
	extra memory	133	152	48	35	1.31	2.55	6.70
	strong scaling	3507	2024	541	410	1.31	2.22	3.22
	limited memory	989	672	399	194	1.42	1.7	2.27
	extra memory	122	77	77	29	1.35	1.76	2.8
	strong scaling	134	68	10	7	1.21	4.02	12.81
	limited memory	47	101	26	8	1.31	2.07	3.41
	extra memory	15	15	10	3	1.5	2.29	3.59
overall						1.07	2.17	12.81

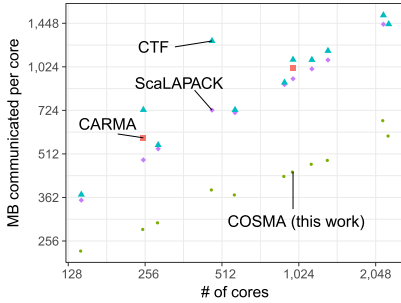
Table 3.4: Average communication volume per MPI rank and measured speedup of COSMA vs the second-best algorithm across all core counts for each of the scenarios.



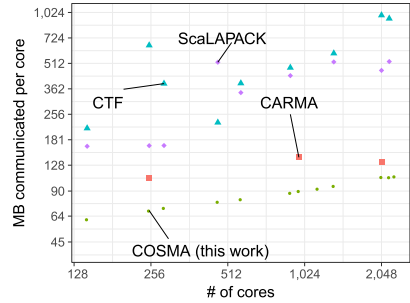
(a) Strong scaling, $N = M = K = 16,384$



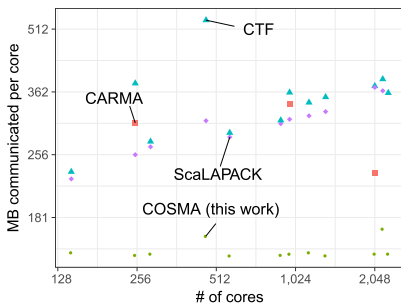
(b) Strong scaling, $N = M = 17,408, K = 3,735,52$



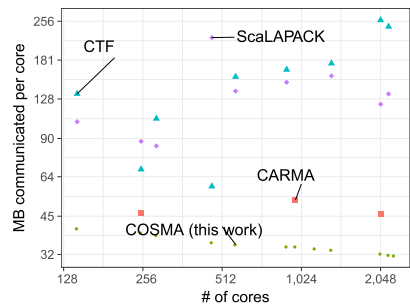
(c) Limited memory, $N = M = K = \sqrt{\frac{pS}{3}}$



(d) Limited memory, $M = N = 979p^{\frac{1}{3}}, K = 1.184p^{\frac{2}{3}}$



(e) Extra memory, $N = M = K = \sqrt{\frac{p^2/3S}{3}}$



(f) Extra memory, $M = N = 979p^{\frac{2}{3}}, K = 1.184p^{\frac{1}{3}}$

Figure 3.6: Total communication volume per core carried out by COSMA, CTF, ScaLAPACK and CARMA for square and “largeK” matrices, as measured by the mpiP profiler.

Summary and Overall Speedups

As discussed in Section 3.8, we evaluate three benchmarks – strong scaling, “limited memory” (no redundant copies of the input are possible), and “extra memory” ($p^{1/3}$ extra copies of the input can fit into combined memory of all cores). Each of them we test for square, “largeK”, “largeM”, and , “flat” matrices, giving twelve cases in total. In Table 3.4, we present arithmetic mean of total communication volume per MPI rank across all core counts. We also report the summary of minimum, geometric mean, and maximum speedups vs the second best-performing algorithm. **Communication**

Volume

As analyzed in Sections 3.5 and 3.6, COSMA reaches I/O lower bound (up to the factor of $\sqrt{S}/(\sqrt{S}+1-1)$). Moreover, optimizations presented in Section 3.7 secure further improvements compared to other state-of-the-art algorithms. In all cases, COSMA performs least communication. Total communication volume for square and “largeK” scenarios is shown in Figure 3.6.

Square Matrices

Figure 3.7 presents the % of achieved peak hardware performance and total runtime for square matrices in all three scenarios. As COSMA is based on the near optimal schedule, it achieves the highest performance *in all cases*. Moreover, its performance pattern is the most stable: when the number of cores is not a power of two, the performance does not vary much compared to all remaining three implementations. We note that matrix dimensions in the strong scaling scenarios ($M = N = K = 2^{14}$) are very small for distributed setting. Yet even in this case COSMA maintains relatively high performance for large numbers of cores: using 4k cores it achieves 35% of peak performance, compared to <5% of CTF and ScaLAPACK, showing excellent strong scaling characteristics.

Tall and Skinny Matrices

Figure 3.8 presents the results for “largeK” matrices. For strong scaling, the minimum number of cores is 2048 (otherwise, the matrices of size $M = N = 17,408$, $K = 3,735,552$ do not fit into memory). Again, COSMA shows the most stable performance with a varying number of cores.

“Flat” Matrices

Matrix dimensions for strong scaling are set to $M = N = 2^{17} = 131,072$ and $K = 2^9 = 512$. Our weak scaling scenario models the rank-K update kernel, with fixed $K = 256$, and $M = N$ scaling accordingly for the “limited”

and “extra” memory cases. Such kernels take most of the execution time in, e.g., matrix factorization algorithms, where updating Schur complements is performed as a rank- K gemm operation [47].

Unfavorable Number of Processors

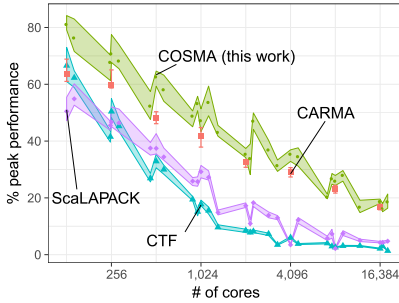
Due to the processor grid optimization (Section 3.7.1), the performance is stable and does not suffer from unfavorable combinations of parameters. E.g., the runtime of COSMA for square matrices $M = N = K = 16,384$ on $p_1 = 9,216 = 2^{10} \cdot 3^2$ cores is 142 ms. Adding an extra core ($p_2 = 9,217 = 13 \cdot 709$), does not change COSMA’s runtime, as the optimal decomposition does not utilize it. On the other hand, CTF for p_1 runs in 600 ms, while for p_2 the runtime *increases* to 1613 ms due to a non-optimal processor decomposition.

Communication-Computation Breakdown

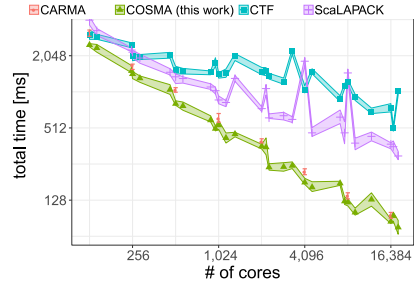
In Figure 3.9 we present the total runtime breakdown of COSMA into communication and computation routines. Combined with the comparison of communication volumes (Figure 3.6, Table 3.4) we see the importance of our I/O optimizations for distributed setting even for traditionally compute-bound MMM. E.g., for square or “flat” matrix and 16k cores, COSMA communicates more than two times less than the second-best (CARMA). Assuming constant time-per-MB, COSMA would be 40% slower if it communicated that much, being slower than CARMA by 30%. For “largeK”, the situation is even more extreme, with COSMA suffering 2.3 times slowdown if communicating as much as the second-best algorithm, CTF, which communicates 10 times more.

Detailed Statistical Analysis

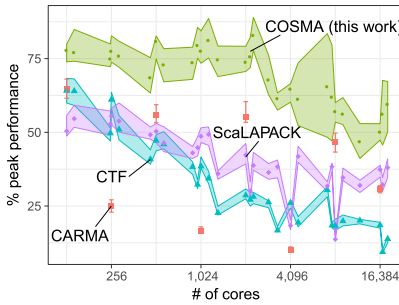
Figure 3.10 provides a distribution of the achieved peak performance across all numbers of cores for all six scenarios. It can be seen that, for example, in the strong scaling scenario and square matrices, COSMA is comparable to the other implementations (especially CARMA). However, for tall-and-skinny matrices with limited memory available, *COSMA lowest achieved performance is higher than the best performance of CTF and ScaLAPACK.*



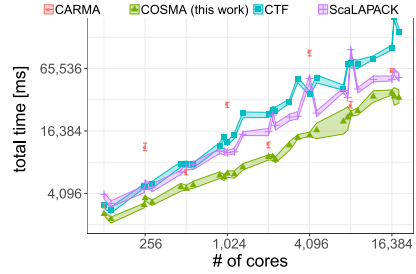
(a) Strong scaling, $N = M = K = 16,384$, % of peak performance



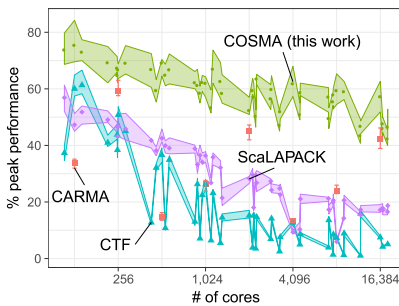
(b) Strong scaling, $N = M = K = 16,384$, total runtime



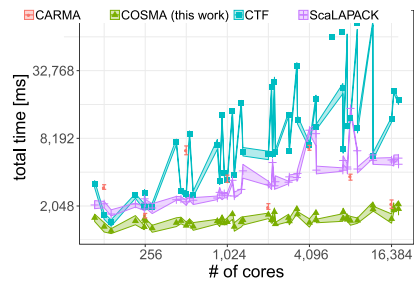
(c) Limited memory, $N = M = K = \sqrt{\frac{pS}{3}}$, % of peak performance



(d) Limited memory, $N = M = K = \sqrt{\frac{pS}{3}}$, total runtime

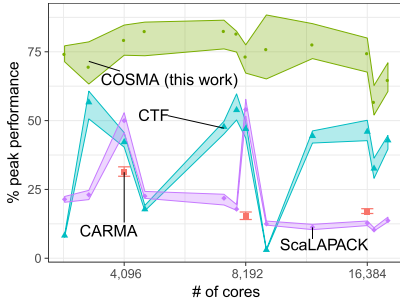


(e) Extra memory, $M = N = K = \sqrt{\frac{p^2/3S}{3}}$, % of peak performance

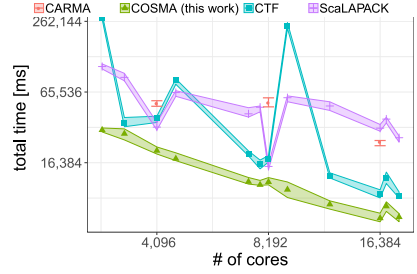


(f) Extra memory, $M = N = K = \sqrt{\frac{p^2/3S}{3}}$, total runtime

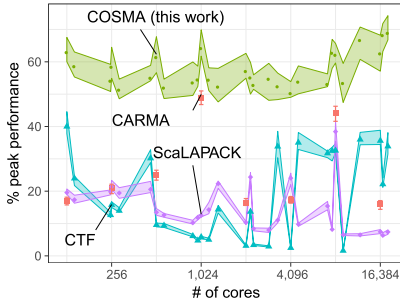
Figure 3.7: Achieved % of peak performance and total runtime by COSMA, CTF, ScaLAPACK and CARMA for square matrices, strong and weak scaling. We show median and 95% confidence intervals.



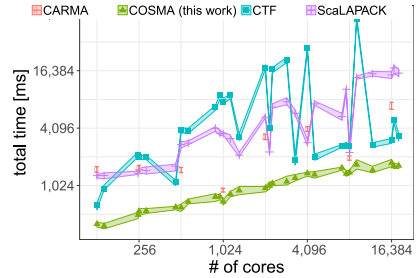
(a) Strong scaling, $N = M = 17,408$, $K = 3,735,552$, % of peak performance



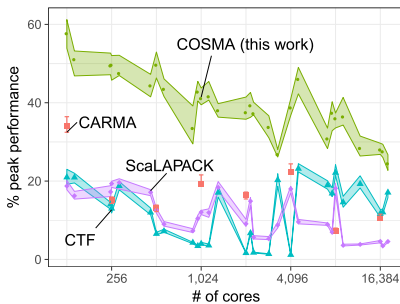
(b) Strong scaling, $N = M = 17,408$, $K = 3,735,552$, total runtime



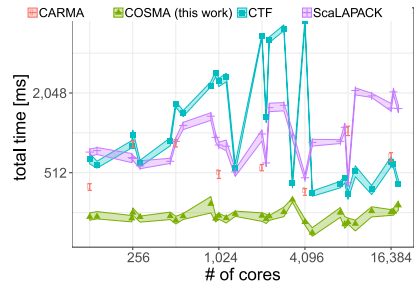
(c) Limited memory, $M = N = 979p^{1/3}$, $K = 1.184p^{2/3}$, % of peak performance



(d) Limited memory, $M = N = 979p^{1/3}$, $K = 1.184p^{2/3}$, total runtime



(e) Extra memory, $M = N = 979p^{2/9}$, $K = 1.184p^{4/9}$, % of peak performance



(f) Extra memory, $M = N = 979p^{2/9}$, $K = 1.184p^{4/9}$, total runtime

Figure 3.8: Achieved % of peak performance and total runtime by COSMA, CTF, ScaLAPACK and CARMA for “largeK” matrices. We show median and 95% confidence intervals.

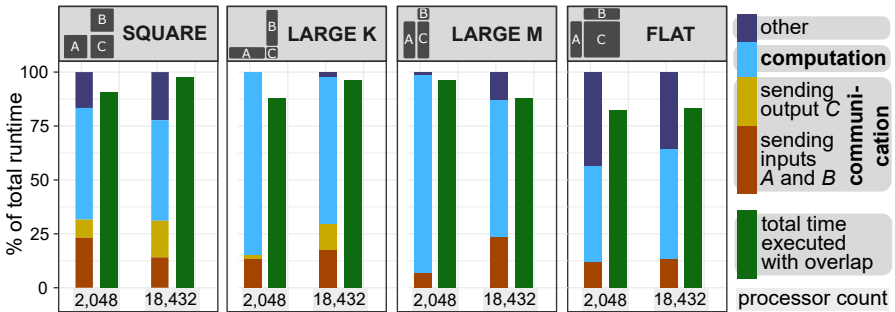


Figure 3.9: Time distribution of COSMA communication and computation kernels for strong scaling executed on the smallest and the largest core counts for each of the matrix shapes. Left bar: no communication–computation overlap. Right bar: overlap enabled.

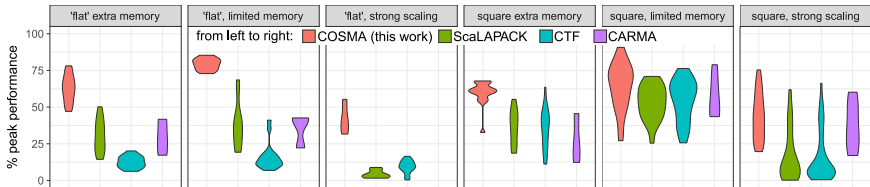


Figure 3.10: Distribution of achieved % of peak performance of the algorithms across all number of cores for “flat” and square matrices.

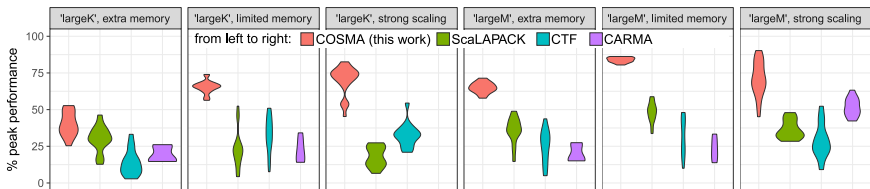


Figure 3.11: Distribution of achieved % of peak performance of the algorithms across all number of cores for tall-and-skinny matrices.

3.10 RELATED WORK

Works on data movement minimization may be divided into two categories: applicable across memory hierarchy (vertical, also called I/O minimization), or between parallel processors (horizontal, also called communication minimization). Even though they are “two sides of the same coin”, in literature they are often treated as separate topics. In our work we combine them: analyze trade-offs between communication optimal (distributed memory) and I/O optimal schedule (shared memory).

3.10.1 General I/O Lower Bounds

Hong and Kung [22] analyzed the I/O complexity for general CDAGs in their the red-blue pebble game, on which we base our work. As a special case, they derived an asymptotic bound $\Omega\left(N^3/\sqrt{S}\right)$ for MMM. Elango et al. [24] extended this work to the red-blue-white game and Liu and Terman [130] proved that it is also P-SPACE complete. Irony et al. [51] extended the MMM lower bound result to a parallel machine with p processors, each having a fast private memory of size S , proving the $\frac{N^3}{2\sqrt{2p\sqrt{S}}} - S$ lower bound on the communication volume per processor. Chan [135] studied different variants of pebble games in the context of memory space and parallel time. Aggarwal and Vitter [23] introduced a two-memory machine that models a blocked access and latency in an external storage. Arge et al. [125] extended this model to a parallel machine. Solomonik et al. [37] combined the communication, synchronization, and computation in their general cost model and applied it to several linear algebra algorithms. Smith and van de Geijn [56] derived a sequential lower bound $2MNK/\sqrt{S} - 2S$ for MMM. They showed that the leading factor $2MNK/\sqrt{S}$ is tight. We improve this result by 1) improving an additive factor of $2S$, but more importantly 2) generalizing the bound to a parallel machine. Our work uses a simplified model, not taking into account the memory block size, as in the external memory model, nor the cost of computation. We motivate it by assuming that the block size is significantly smaller than the input size, the data is layout contiguously in the memory, and that the computation is evenly distributed among processors.

3.10.2 *Shared Memory Optimizations*

I/O optimization for linear algebra includes such techniques as loop tiling and skewing [48], interchanging and reversal [136]. For programs with multiple loop nests, Kennedy and McKinley [137] showed various techniques for loop fusion and proved that in general this problem is NP-hard. Later, Darte [138] identified cases when this problem has polynomial complexity.

Toledo [139] in his survey on Out-Of-Core (OOC) algorithms analyzed various I/O minimizing techniques for dense and sparse matrices. Mohanty [140] in his thesis optimized several OOC algorithms. Irony et al. [51] proved the I/O lower bound of classical MMM on a parallel machine. Ballard et al. [12] proved analogous results for Strassen's algorithm. This analysis was extended by Scott et al. [141] to a general class of Strassen-like algorithms.

Although we consider only dense matrices, there is an extensive literature on sparse matrix I/O optimizations. Bender et al. [142] extended Aggarwal's external memory model [23] and showed I/O complexity of the sparse matrix-vector (SpMV) multiplication. Greiner [143] extended those results and provided I/O complexities of other sparse computations.

3.10.3 *Distributed Memory Optimizations*

Distributed algorithms for dense matrix multiplication date back to the work of Cannon [60], which has been analyzed and extended many times [144] [126]. In the presence of extra memory, Aggarwal et al. [63] included parallelization in the third dimension. Solomonik and Demmel [52] extended this scheme with their 2.5D decomposition to arbitrary range of the available memory, effectively interpolating between Cannon's 2D and Aggarwal's 3D scheme. A recursive, memory-oblivious MMM algorithm was introduced by Blumofe et al. [145] and extended to rectangular matrices by Frigo et al. [146]. Demmel et al. [64] introduced CARMA algorithm which achieves the asymptotic complexity for all matrix and memory sizes. We compare COSMA with these algorithms, showing that we achieve better results both in terms of communication complexity and the actual runtime performance. Lazzaro et al. [147] used the 2.5D technique for sparse matrices, both for square and rectangular grids. Koanantakool et al. [148] observed that for sparse-dense MMM, 1.5D decomposition performs less communication than 2D and 2.5D schemes, as it distributes only the sparse matrix.

3.11 SUMMARY

In this chapter we present a new method (Lemma 3.3) for assessing tight I/O lower bounds of algorithms using their CDAG representation and the red-blue pebble game abstraction. As a use case, we prove a tight bound for MMM, both for a sequential (Theorem 3.1) and parallel (Theorem 3.2) execution. Furthermore, our proofs are constructive: our COSMA algorithm is near I/O optimal (up to the factor of $\frac{\sqrt{S}}{\sqrt{S+1}-1}$, which is less than 0.04% from the lower bound for 10MB of fast memory) for any combination of matrix dimensions, number of processors and memory sizes. This is in contrast with the current state-of-the-art algorithms, which are communication-inefficient in some scenarios.

To further increase the performance, we introduce a series of optimizations, both on an algorithmic level (processor grid optimization (Section 3.7.1) and blocked data layout (Section 3.7.6)) and hardware-related (enhanced communication pattern (Section 3.7.2), communication-computation overlap (Section 3.7.3), one-sided (Section 3.7.4) communication). The experiments confirm the superiority of COSMA over the other analyzed algorithms - our algorithm significantly reduces communication in all tested scenarios, supporting our theoretical analysis. Most importantly, our work is of practical importance, being maintained as an open-source implementation and achieving a time-to-solution speedup of up to 12.8x times compared to highly optimized state-of-the-art libraries.

The important feature of our method is that it does not require any manual parameter tuning and is generalizable to other machine models (e.g., multiple levels of memory) and linear algebra kernels. In the following chapter, we extend this analysis to LU or Cholesky factorizations. We believe that the “bottom-up” approach will lead to developing more efficient distributed algorithms in the future.

I/O OPTIMAL MATRIX FACTORIZATIONS

This chapter presents our work published at the SC'21 conference [149]. It is a continuation of the previous Chapter, both in terms of theory (extending the X-Partitioning to a general class of programs), as well as practice (matrix multiplication is one of the building blocks of matrix factorizations). The CONfLUX library was developed by me, Marko Kabić from CSCS, Alexandros Nikolaos Ziogas, and two of my students: André Gaillard and Jens Eirik Saethre.

4.1 INTRODUCTION

Matrix factorizations, such as LU and Cholesky decompositions, play a crucial role in many scientific computations [104, 121, 150], and their performance can dominate the overall runtime of entire applications [151]. Therefore, accelerating these routines is of great significance for numerous domains [40, 68]. The ubiquity and importance of LU factorization is even reflected by the fact that it is used to rank top supercomputers worldwide [152].

Since the arithmetic complexity of matrix factorizations is $\mathcal{O}(N^3)$ while the input size is $\mathcal{O}(N^2)$, these kernels are traditionally considered compute-bound. However, the end of Dennard scaling [26] puts increasing pressure on data movement minimization, as the cost of moving data far exceeds its computation cost, both in terms of power and time [153, 154]. Thus, deriving algorithmic I/O lower bounds is a subject of both theoretical analysis [22, 51, 57] and practical value for developing I/O-efficient schedules [37, 113, 155].

While asymptotically optimal matrix factorizations were proposed, among others, by Ballard et al. [53] and Solomonik et al. [52, 155], we observe two major challenges with the existing approaches: First, the presented algorithms are only asymptotically optimal: the I/O cost of these proposed parallel algorithms can be as high as 7 times the lower bound for LU [52] and up to 16 times for Cholesky [155]. This means that they communicate less than “standard” 2D algorithms like ScaLAPACK [127] only for almost prohibitively large numbers of processors — e.g., according

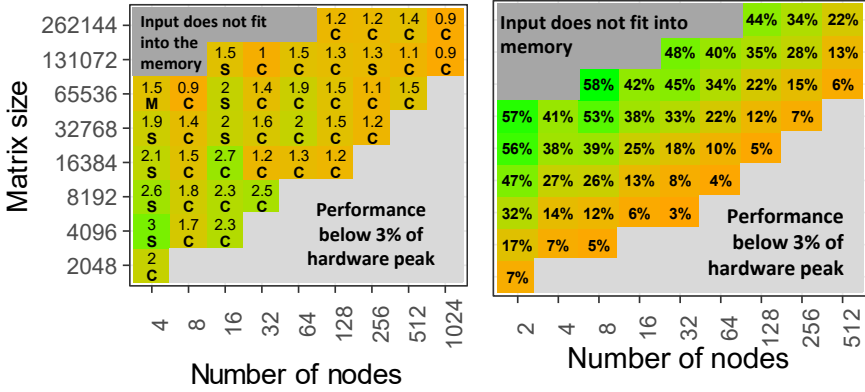


Figure 4.1: Left: measured runtime speedup of CONfLUX vs. fastest state-of-the-art library (S=SLATE [156], C=CANDMC [157], S=MKL [158]). Right: CONfLUX's achieved % of machine peak performance.

to the LU cost model [52], it requires more than 15,000 processors to communicate less than an optimized 2D algorithm. Second, their time-to-solution performance can be worse than highly-optimized, existing 2D-parallel libraries [155].

To tackle these challenges, we first provide a *general* method for deriving *precise* I/O lower bounds of Disjoint Array Access Programs (DAAP) — a broad range of programs composed of a sequence of statements enclosed in an arbitrary number of nested loops. We then illustrate the applicability of our framework to derive parallel I/O lower bounds of Cholesky and LU factorizations: $\frac{1}{3} \frac{N^3}{p\sqrt{S}}$ and $\frac{2}{3} \frac{N^3}{p\sqrt{S}}$ elements, respectively, where N is the matrix size, p is the number of processors, and S is the local memory size.

Moreover, we use the insights from deriving the above lower bounds to develop CONfLUX and CONfCHOX, near communication-optimal parallel LU and Cholesky factorization algorithms that minimize data movement across the 2.5D processor decomposition. For LU factorization, to further reduce the latency and bandwidth cost, we use a row-masking tournament pivoting strategy resulting in a communication requirement of $\frac{N^3}{p\sqrt{S}} + \mathcal{O}(\frac{N^2}{p})$ elements per processor, where the leading order term is only 1.5 times the lower bound. Furthermore, to secure high performance, we carefully tune block sizes and communication routines to maximize the efficiency

of local computations such as `trsm` (triangular solve) and `gemm` (matrix multiplication).

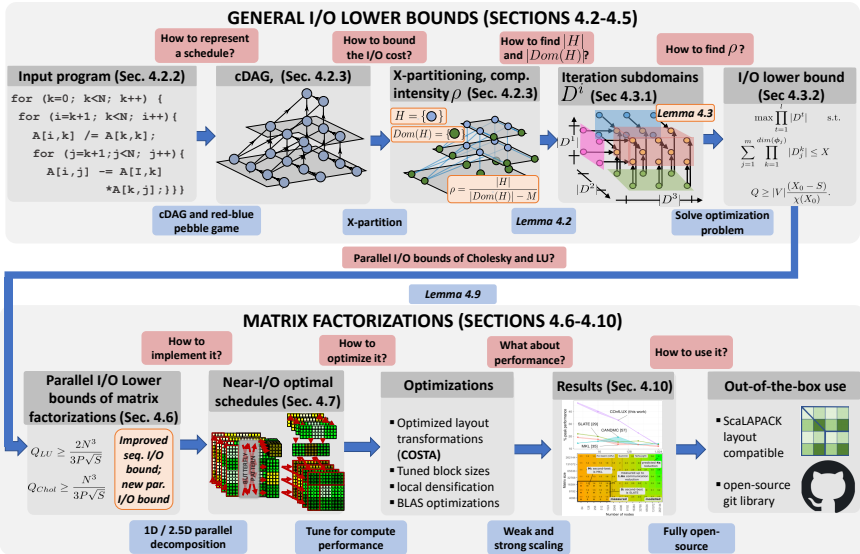


Figure 4.2: From the input program through the I/O lower bounds to communication-minimizing parallel schedules and high performing implementations. In this paper, we mainly focus on the Cholesky and LU factorizations. The proofs of the lemmas presented in this work can be found in the AD/AE appendix.

We measure both communication volume and achieved performance of `ConfLUX` and `ConfCHOX` and compare them to state-of-the-art libraries: a vendor-optimized Intel MKL [158], SLATE [156] (a recent library targeting exascale systems), as well as CANDMC [157, 159] and CAPITAL [155, 160] (codes based on the asymptotically optimal 2.5D decomposition). In our experiments on the Piz Daint supercomputer, we measure up to 1.6x communication reduction compared to the second-best implementation. Furthermore, our 2.5D decomposition communicates asymptotically less than SLATE and MKL, with even greater expected benefits on exascale machines. Compared to the communication-avoiding CANDMC library with I/O cost of $5N^3 / (p\sqrt{S})$ elements [52], `ConfLUX` communicates five times less. Most importantly, *our implementations outperform all compared libraries in almost all scenarios*, both for strong and weak scaling, reducing

the time-to-solution by up to three times compared to the second best performing library (Figure 4.1).

In this work, we make the following contributions:

- A general method for deriving parallel I/O lower bounds of a broad range of linear algebra kernels.
- *CONfLUX* and *CONfCHOX*, provably near-I/O-optimal parallel algorithms for LU and Cholesky factorizations, with their full communication volume analysis.
- Open-source and fully ScaLAPACK-compatible implementations of our algorithms that outperform existing state-of-the-art libraries in almost all scenarios.

A bird’s eye view of our work is presented in Figure 4.2.

4.2 BACKGROUND

We now establish the background for our theoretical model (Sections 4.3–4.5). We use it to derive parallel I/O lower bounds for Cholesky and LU factorizations (Section 4.6) that will guide the design of our communication-minimizing implementations (Section 4.7).

4.2.1 *Machine Model*

We use similar machine model to the one introduced in Chapter 3. Here we briefly outline its key features.

Sequential machine. A computation is performed on a sequential machine with a fast memory of limited size and unlimited slow memory. The fast memory can hold up to S elements at any given time. To perform any computation, all input elements must reside in fast memory, and the result is stored in fast memory.

Parallel machine. The sequential model is extended to a machine with p processors, each equipped with a private fast memory of size S . There is no global memory of unlimited size — instead, elements are transferred between processors’ fast memories.

4.2.2 Input Programs

We consider a general class of programs that operate on multidimensional arrays. Array elements can be loaded from slow to fast memory, stored from fast to slow memory, and computed inside fast memory. These elements have *versions* that are incremented every time they are updated. We model the program execution as a computational directed acyclic graph (CDAG, details in Section 4.2.3), where each vertex corresponds to a different version of an array element. Thus, for a statement $A[i, j] \leftarrow f(A[i, j])$, a vertex corresponding to $A[i, j]$ after applying f is different from a vertex corresponding to $A[i, j]$ before applying f . In a CDAG, this is expressed as an edge from vertex $A[i, j]$ before f to vertex $A[i, j]$ after f . Initial versions of each element do not have any incoming edges and thus form the CDAG inputs. *The distinction between elements and vertices* is important for our I/O lower bounds analysis, as we will investigate how many vertices are computed for a given number of loaded vertices.

An input program is a collection of statements S enclosed in loop nests, each of the following form (we use the loop nest notation introduced by Dinh and Demmel [49]):

$$\text{for } \psi^1 \in \mathcal{D}^1, \text{ for } \psi^2 \in \mathcal{D}^2(\psi^1), \dots, \text{ for } \psi^l \in \mathcal{D}^l(\psi^1, \dots, \psi^{l-1}) : \\ S : A_0[\boldsymbol{\phi}_0(\boldsymbol{\psi})] \leftarrow f(A_1[\boldsymbol{\phi}_1(\boldsymbol{\psi})], A_2[\boldsymbol{\phi}_2(\boldsymbol{\psi})], \dots, A_m[\boldsymbol{\phi}_m(\boldsymbol{\psi})]),$$

where (cf. Figure 4.3 for a summary) for each innermost loop iteration, statement S is an evaluation of some function f on m inputs, where every input is an element of array $A_j, j = 1, \dots, m$, and the result of f is stored to the output array A_0 .

Each loop has an associated *iteration variable* ψ^t that iterates over its domain $\psi^t \in \mathcal{D}^t$. All l iteration variables form the *iteration vector* $\boldsymbol{\psi} = [\psi^1, \dots, \psi^l]$. Array elements are accessed by an *access function vector* $\boldsymbol{\phi}_j = [\phi_j^1, \dots, \phi_j^{\dim(A_j)}]$ that maps $\dim(A_j)$ iteration variables to a *unique* element in array A_j (note that the access function vector is injective). Only vertices associated with the newest element versions can be referenced. Furthermore, a given vertex can be referenced by only one access function vector per statement. We refer to this as the *disjoint access property*. The *access dimension* of $A_j(\boldsymbol{\phi}_j)$, denoted $\dim(A_j(\boldsymbol{\phi}_j))$, is the number of different iteration variables present in $\boldsymbol{\phi}_j$. We call such programs Disjoint Access Array Programs.

Example: Consider statement S1 of LU factorization (Figure 4.3). The loop nest depth is $l = 2$, with two iteration variables $\psi^1 = k$ and $\psi^2 = i$ forming the iteration vector $\boldsymbol{\psi} = [k, i]$. For access $A[k, k]$, the access function vector $\boldsymbol{\phi}_j = [k, k]$ is a function of only one iteration variable k . Therefore, $\dim(A_j) = 2$, but $\dim(A_j(\boldsymbol{\phi}_j)) = 1$.

4.2.3 I/O Complexity and Pebble Games

We now establish the relationship between DAAP and the X-Partitioning abstraction introduced in Chapter 3. For completeness, we briefly sketch key features of the red-blue pebble game [22], dominator and minimum sets, as well as the X-Partitioning itself. For complete description, we refer readers to Chapter 3.

4.2.3.1 CDAG and the red-blue pebble game

Every vertex $v \in V$ represents the result of a unique computation stored in some memory, and a directed edge $(u, v) \in E$ represents a data dependency. Vertices without any incoming (outgoing) edges are called *inputs (outputs)*. To perform a computation, i.e., to evaluate the value corresponding to vertex v , all vertices that are direct predecessors of v must be loaded into fast memory. The vertices that are currently in fast memory are marked by a red pebble on the corresponding vertex of the CDAG. Since the size of fast memory is limited, we can never have more than S red pebbles on the CDAG at any moment. Analogously, the contents of the slow memory (of unlimited size) is represented by an unlimited number of blue pebbles.

4.2.3.2 Dominator and Minimum Sets

For any subset of vertices $\mathcal{H} \subset V$, a *dominator set* $Dom(\mathcal{H})$ is a set such that every path in the CDAG from an input vertex to any vertex in \mathcal{H} must contain at least one vertex in $Dom(\mathcal{H})$. In general, for a given \mathcal{H} , its $Dom(\mathcal{H})$ is not uniquely defined. The *minimum set* $Min(\mathcal{H})$ is the set of all vertices in \mathcal{H} that do not have any immediate successors in \mathcal{H} . In this work, to avoid the ambiguity of non-uniqueness of dominator set size (in principle, for any subset, its valid dominator set is always the whole V), we will refer to $Dom_{min}(\mathcal{H})$ as a minimum dominator set, i.e. a dominator set with the smallest size.

Intuition. One can think of \mathcal{H} 's dominator set as a set of inputs required to execute subcomputation \mathcal{H} , and of \mathcal{H} 's minimum set as the output of \mathcal{H} . We use the notions of $Dom_{min}(\mathcal{H})$ and $Min(\mathcal{H})$ when proving I/O lower bounds. Intuitively, we bound computation "volume" (number of vertices in \mathcal{H}) by its communication "surface", comprised of its inputs - vertices in $Dom_{min}(\mathcal{H})$ - and outputs - vertices in $Min(\mathcal{H})$.

4.2.3.3 X-Partitioning

Introduced in Chapter 3, X-Partitioning generalizes the S-partitioning abstraction [22]. An X-partition of a CDAG is a collection of s mutually disjoint subsets (referred to as *subcomputations*) $\mathcal{S}(X) = \{\mathcal{H}_1, \dots, \mathcal{H}_s\}$, $\cup_{i=1}^s \mathcal{H}_i = V$ with two additional properties:

- $\mathcal{S}(X)$ has no cyclic dependencies between subcomputations.
- $\forall \mathcal{H}_i, |Dom_{min}(\mathcal{H}_i)| \leq X$ and $|Min(\mathcal{H}_i)| \leq X$.

For a given CDAG and for any given $X > S$, let $\Pi(X)$ denote a set of all its valid X-partitions, $\mathcal{S}(X) \in \Pi(X)$. In Chapter 3 we prove that an I/O optimal schedule of G that performs Q load and store operations has an associated X-partition $\mathcal{S}_{opt}(X) \in \Pi(X)$ with size $|\mathcal{S}_{opt}(X)| \leq \frac{Q+X-S}{X-S}$ for any $X > S$ (Lemma 3.3).

4.2.3.4 Deriving lower bounds

To bound the I/O cost, we further need the *computational intensity* ρ . For each subcomputation \mathcal{H}_i , ρ_i is defined as a ratio of the number of computations (vertices) in \mathcal{H}_i to the number of I/O operations required to pebble \mathcal{H}_i , where the latter is bounded by the size of the dominator set $Dom(\mathcal{H}_i)$.

Observe that Lemma 3.4 requires the sizes of the maximum reuse set $R(S)$ and the minimum I/O set $T(S)$. Knowing them in advance may be infeasible in the general case. However, we can bound them by $R(S) \leq S$ and $T(S) \geq 0$. Substituting these bounds to Lemma 3.4, we obtain the following result:

Lemma 4.1. For any constant X_c , the number of I/O operations Q required to pebble a CDAG $G = (V, E)$ with $|V| = n$ vertices using S red pebbles is bounded by $Q \geq n/\rho$, where $\rho = \frac{|\mathcal{H}_{max}|}{X_c - S}$ is the maximal computational intensity and $\mathcal{H}_{max} = \arg \max_{\mathcal{H} \in \mathcal{S}(X_c)} |\mathcal{H}|$ is the largest subcomputation among all valid X_c -partitions.

4.3 GENERAL SEQUENTIAL I/O LOWER BOUNDS

We now present our method for deriving the I/O lower bounds of a sequential execution of programs in the form defined in Section 4.2.2. Specifically, in Section 4.3.2 we derive I/O bounds for programs that contain only a single statement. In Section 4.4 we extend our analysis to capture interactions and reuse between multiple statements.

We start by stating our key lemma:

Lemma 4.2. *If $|H_{max}|$ can be expressed as a closed-form function of X , that is if there exists some function χ such that $|H_{max}| = \chi(X)$, then the lower bound on Q can be expressed as*

$$Q \geq n \frac{(X_0 - S)}{\chi(X_0)},$$

where $X_0 = \arg \min_X \rho = \arg \min_X \frac{\chi(X)}{X-S}$.

Intuition. $\chi(X)$ expresses computation “volume”, while X is its input “surface”. The term $X - S$ bounds the required communication and it comes from the fact that not all inputs have to be loaded (at most S of them can be reused). X_0 corresponds to the situation where the ratio of this “volume” to the required communication is minimized (corresponding to a highest lower bound).

Proof. Note that Lemma 4.1 is valid for any X_c (i.e., for any X_c , it gives a valid lower bound). Yet, these bounds are not necessarily tight. As we want to find tight I/O lower bounds, we need to maximize the lower bound. X_0 by definition minimizes ρ ; thus, it maximizes the bound. Lemma 4.2 then follows directly from Lemma 4.1 by substituting $\rho = \frac{\chi(X_0)}{X_0-S}$. \square

Note. If function $\chi(X)$ is differentiable and has a global minimum, we can find X_0 by, e.g., solving the equation $\frac{d\frac{\chi(X)}{X-S}}{dX} = 0$. The key limitation is that it is not always possible to find χ , that is, to express $|H_{max}|$ solely as a function of X . However, for many linear algebra kernels $\chi(X)$ exists. Furthermore, one can relax this problem preserving the correctness of the lower bound, that is, by finding a function $\hat{\chi} : \forall_X \hat{\chi}(X) \geq \chi(X)$.

To find $\chi(X)$, we take advantage of the DAAP structure. Observe that every computation (and therefore, every compute vertex $v \in V$ in the CDAG $G = (V, E)$) is executed in a different iteration of the loop nest, and thus, there is a one-to-one mapping from a value of the iteration vector ψ to the compute vertex v . Moreover, each vertex accessed from any of the input

arrays A_i is also associated with some iteration vector value - however, if $\dim(A_i) < l$, this is a one-to-many relation, as the same input vertex may be used to evaluate multiple compute vertices v . This is, in fact, the source of the data reuse, and exploiting this relation is a key to minimizing the I/O cost. If for all input arrays A_i we have that $\dim(A_i) = l$, then for each compute vertex v , m different, unique input vertices are required, there is no data reuse and it implies a trivial computational intensity $\rho = \frac{1}{m}$.

The high-level idea of our method is to *count how many different iteration vector values ϕ can be formed if we know how many different values each iteration variable ϕ^1, \dots, ϕ^l takes*. We now formalize this in Lemmas 4.3-4.8.

4.3.1 Iteration vector, iteration domain, access set

Each execution of statement S is associated with the *iteration vector* value $\psi = [\psi^1, \dots, \psi^l] \in \mathbb{N}^l$ representing the current iteration, that is, the values of iteration variables ψ^1, \dots, ψ^l . Each subcomputation H is uniquely defined by all iteration vectors' values associated with vertices pebbled in $H = \{\psi_1, \dots, \psi_{|H|}\}$. For each iteration variable ψ^t , $t = 1, \dots, l$, denote the set of all values that ψ^t takes during H as D^t . We define $\mathbf{D} = [D^1, \dots, D^l] \subseteq \mathcal{D}$ as the *iteration domain* of subcomputation H .

Furthermore, recall that each input access $A_j[\phi_j(\psi)]$ is uniquely defined by $\dim(\phi_j)$ iteration variables $\psi_j^1, \dots, \psi_j^{\dim(\phi_j)}$. Denote the set of all values each of ψ_j^k takes during H as D_j^k . Given \mathbf{D} , we also denote the number of different vertices that are accessed from each input array A_j as $|A_j(\mathbf{D})|$.

We now state the lemma which bounds $|H|$ by the iteration sets' sizes $|D^t|$ (the intuition behind the lemma is depicted in Figure 4.4):

Lemma 4.3. *Given the ranges of all iteration variables D^t , $t = 1, \dots, l$ during subcomputation H , if $|H| = \prod_{t=1}^l |D^t|$, then $\forall j = 1, \dots, m : |A_j(\mathbf{D})| = \prod_{k=1}^{\dim(\phi_j)} |D_j^k|$ and $|H|$ is maximized among all valid subcomputations that iterate over $\mathbf{D} = [D^1, \dots, D^l]$.*

Intuition. *Lemma 4.3 states that if each iteration variable ψ^t , $t = 1, \dots, l$ takes $|R_h^t|$ different values, then there are at most $\prod_{t=1}^l |D^t|$ different iteration vectors ψ which can be formed in H . So, intuitively, to maximize $|H|$, all combinations of values ψ^t should be evaluated. On the other hand, this also implies maximization of all access sizes $|A_j(\mathbf{D})| = \prod_{k=1}^{\dim(\phi_j)} |D_j^k|$.*

To prove it, we now introduce two auxiliary lemmas:

Lemma 4.4. *For statement S , the size $|H|$ of subcomputation H (number of vertices of S computed during H) is bounded by the sizes of the iteration variables' sets $R_h^t, t = 1, \dots, l$:*

$$|H| \leq \prod_{t=1}^l |D^t|. \quad (4.1)$$

Proof. Inequality 5.2 follows from a combinatorial argument: each computation in H is uniquely defined by its iteration vector $[\psi^1, \dots, \psi^l]$. As each iteration variable ψ^t takes $|R_h^t|$ different values during H , we have $|R_h^1| \cdot |R_h^2| \cdot \dots \cdot |R_h^l| = \prod_{t=1}^l |D^t|$ ways how to uniquely choose the iteration vector in H . \square

Now, given D , we want to assess how many different vertices are accessed for each input array A_j . Recall that this number is denoted as access size $|A_j(D)|$.

We will apply the same combinatorial reasoning to $A_j(D)$. For each access $A_j[\phi_j(\psi)]$, each one of $\psi^k, k = 1, \dots, \dim(\phi_j)$ iteration variables loops over set $R_{h,j}^k$ during subcomputation H . We can thus bound size of $A_j(D)$ similarly to Lemma 5.1:

Lemma 4.5. *The access size $|A_j(D)|$ of subcomputation H (the number of vertices from the array A_j required to compute H) is bounded by the sizes of $\dim(\phi_j)$ iteration variables' sets $R_{h,j}^k, k = 1, \dots, \dim(\phi_j)$:*

$$\forall_{j=1, \dots, m} : |A_j(D)| \leq \prod_{k=1}^{\dim(\phi_j)} |D_j^k| \quad (4.2)$$

where $D_j^k \ni \psi_j^k$ is the set over which iteration variable ψ_j^k iterates during H .

Proof. We use the same combinatorial argument as in Lemma 5.1. Each vertex in $A_j(D)$ is uniquely defined by $[\psi_j^1, \dots, \psi_j^{\dim(\phi_j)}]$. Knowing the number of different values each ψ_j^k takes, we bound the number of different access vectors $\phi_j(\psi_h)$. \square

Example: Consider once more statement $S1$ from LU factorization in Figure 4.3. We have $\phi_0 = [i, k]$, $\phi_1 = [i, k]$, and $\phi_2 = [k, k]$. Denote the iteration subdomain for subcomputation H as $D = \{[k^1, i^1], \dots, [k^{|H|}, i^{|H|}]\}$, where each variable k and i iterates over its set $k^g \in \{\psi_{k,1}, \dots, \psi_{k,K}\} = D^k$ and $i^g \in \{\psi_{i,1}, \dots, \psi_{i,I}\} = D^i$, for $g = 1, \dots, |H|$. Denote the sizes of these sets as $|D^k| = K$ and $|D^i| = I$, that is, during H , variable k takes K different values and i takes I different values. For ϕ_1 ,

both iteration variables used are different: k and i . Therefore, we have (Equation 5.3) $|A_1(\mathbf{D})| \leq K_h \cdot I_h$. On the other hand, for ϕ_2 , the iteration variable k is used twice. Recall that the access dimension is the minimum number of different iteration variables that uniquely address it (Section 4.2.2), so its dimension is $\dim(A_2) = 1$ and the only iteration variable needed to uniquely determine ϕ_2 is k . Therefore, $|A_2(\mathbf{D})| \leq K_h$.

Dominator set. Input vertices A_1, \dots, A_m form a dominator set of vertices A_0 , because any path from graph inputs to any vertex in A_0 must include at least one vertex from A_1, \dots, A_m . This is also the *minimum* dominator set, because of the disjoint access property (Section 4.2.2): any path from graph inputs to any vertex in A_0 can include at most one vertex from A_1, \dots, A_m .

Proof of Lemma 4.3. For subcomputation H , we have $|\bigcup_{j=1}^m A_j(\mathbf{D})| \leq X$ (by the definition of an X -partition). Again, by the disjoint access property, we have $\forall j_1 \neq j_2 : A_{j_1}(\mathbf{D}) \cap A_{j_2}(\mathbf{D}) = \emptyset$. Therefore, we also have $|\bigcup_{j=1}^m A_j(\mathbf{D})| = \sum_{j=1}^m |A_j(\mathbf{D})|$. We now want to maximize $|H|$, that is to find H_{max} to obtain computational intensity ρ (Lemma 4.2).

Now we prove that to maximize $|H|$, inequalities 5.2 and 5.3 must be tight (become equalities).

From proof of Lemma 5.1 it follows that $|H|$ is maximized when iteration vector ψ takes all possible combinations of iteration variables $\psi^t \in D^t$ during H . But, as we visit each combination of all l iteration variables, for each access A_j every combination of its $[\psi_j^1, \dots, \psi_j^{\dim(\phi_j)}]$ iteration variables is also visited. Therefore, for every $j = 1, \dots, m$, each access size $|A_j(\mathbf{D})|$ is maximized (Lemma 5.2), as access functions are injective, which implies that for each combination of $[\psi_j^1, \dots, \psi_j^{\dim(\phi_j)}]$, there is one access to A_j . $\prod_{t=1}^l |R_h^t|$ is then the upper bound on $|H|$, and its tightness implies that all bounds on access sizes $|A_j(\mathbf{D})| \leq \prod_{k=1}^{\dim(\phi_j)} |D_j^k|$ are also tight. \square

Intuition. Lemma 4.3 states that if each iteration variable ψ^t , $t = 1, \dots, l$ takes $|D^t|$ different values, then there are at most $\prod_{t=1}^l |D^t|$ different iteration vector values ψ that can be formed in H . Thus, to maximize $|H|$ all combinations of values of ψ^t should be evaluated. On the other hand, this also implies the maximization of all access sizes $|A_j(\mathbf{D})| = \prod_{k=1}^{\dim(\phi_j)} |D_j^k|$. This result is more general than, e.g., polyhedral techniques [57, 161, 162] as it does not require loop nests to be affine. Instead, it solely relies on set algebra and combinatorial methods.

4.3.2 Finding the I/O Lower Bound

Denoting $H_{max} = \arg \max_{H \in \mathcal{P}(X)} |H|$ as the largest subcomputation among all valid X -partitions, we use Lemma 4.3 and combine it with the dominator set constraint from Section 4.2.3.3. Note that all access set sizes are strictly positive integers $|D^t| \in \mathbb{N}_+, t = 1, \dots, l$. Otherwise, if any of the sets is empty, no computation can be performed. However, as we only want to find the bound on the number of I/O operations, we relax the integer constraints and replace them with $|D^t| \geq 1$. Then, we formulate finding $\chi(X)$ (Lemma 4.2) as the following optimization problem:

$$\begin{aligned} & \max \prod_{t=1}^l |D^t| \quad \text{s.t.} \\ & \sum_{j=1}^m \prod_{k=1}^{\dim(\phi_j)} |D_j^k| \leq X \\ & \forall 1 \geq t \geq l : |D^t| \geq 1 \end{aligned} \quad (4.3)$$

We then find $|H_{max}| = \chi(X)$ as a function of X using Karush–Kuhn–Tucker (KKT) conditions [163]. Next, we solve

$$\frac{d\chi(X)}{dX} = 0. \quad (4.4)$$

Denoting X_0 as the solution to Equation (4.4), we finally obtain

$$Q \geq |V| \frac{(X_0 - S)}{\chi(X_0)}. \quad (4.5)$$

Computational intensity and out-degree-one vertices. There exist CDAGs where every non-input vertex has at least $u \geq 0$ direct predecessors that are input vertices with out-degree 1. We can use this fact to put an additional bound on the computational intensity.

Lemma 4.6. *If in a CDAG $G = (V, E)$ every non-input vertex has at least u direct predecessors with out-degree one that are graph inputs, then the maximum computational intensity ρ of this CDAG is bounded by $\rho \leq \frac{1}{u}$.*

Proof. By the definition of the red-blue pebble game, all inputs start in slow memory, and therefore, have to be loaded. By the assumption on the CDAG, to compute any non-input vertex $v \in V$, at least u input vertices need to

have red pebbles placed on them using a load operation. Because these vertices do not have any other direct successors (their out-degree is 1), they cannot be used to compute any other non-input vertex w . Therefore, each computation of a non-input vertex requires at least u unique input vertices to be loaded. \square

Example: Consider Figure 4.5. In a), each compute vertex $C[i, j]$ has two input vertices: $A[i, j]$ with out-degree 1, and $b[j]$ with out-degree n , thus $u = 1$. As both array A and vector b start in the slow memory (having blue pebbles on each vertex), for each computed vertex from C , at least one vertex from A has to be loaded, therefore $\rho \leq 1$. In b), each computation needs two out-degree 1 vertices, one from vector a and one from vector b , resulting in $u = 2$. Thus, $\rho \leq \frac{1}{2}$.

4.4 DATA REUSE ACROSS MULTIPLE STATEMENTS

Until now, we have analyzed each statement separately. However, almost all computational kernels contain multiple statements connected by data dependencies — e.g., a column update (S1) and a trailing matrix update (S2) in LU factorization (Figure 4.3). The challenge here is that, in general, I/O cost Q is not composable: due to the data reuse, the total I/O cost of the program may be smaller than the sum of I/O costs of its constituent kernels. In this section we examine how these dependencies influence the total I/O cost of a program.

We derive I/O lower bounds for programs with w statements S_1, \dots, S_w in two steps. First, we analyze each statement S_i separately, as in Section 4.3. Then, we derive how many loads could be avoided at most during one statement if another statement owned shared data. There are two cases where data reuse can occur: **I**) input overlap, where shared arrays are inputs for multiple statements, and **II**) output overlap, where the output array of one statement is the input array of another.

Case I. Assume there are w statements in the program, and there are k arrays $A_j, j = 1, \dots, k$ that are shared between at least two statements. We still evaluate each statement separately, but we will subtract the upper bound on shared loads $Q_{tot} \geq \sum_{i=1}^w Q_i - \sum_{j=1}^k |Reuse(A_j)|$, where $|Reuse(A_j)|$ is the reuse bound on array A_j (Section 4.4.1). **Case II.** Consider each pair of “producer-consumer” statements S and T , that is, the output of S is the input of T . The I/O lower bound Q_S of statement S does not change due to the reuse, as the same number of loads has to be performed to evaluate S . On the other hand, it may invalidate Q_T , as the dominator set of T formu-

lated in Section 4.3.1 may not be minimal — inputs of a statement may not be graph inputs anymore. For each “consumer” statement T , we reevaluate $Q'_T \leq Q_T$ using Lemma 4.8. Finally, for a program consisting of w statements in total, connected by the output overlap, we have $Q_{tot} \geq \sum_{i=1}^w Q'_i$. Note that for each “producer” statement i , $Q'_i = Q_i$, i.e. output overlap does not change their I/O lower bound.

4.4.1 Case I: Input Reuse and Reuse Size

Consider two statements S and T that share one input array A_i . Let $|A_i(\mathbf{R}_S)|$ denote the total number of accesses to A_i during the I/O optimal execution of a program that contains only statement S . Naturally, $|A_i(\mathbf{R}_T)|$ denotes the same for a program containing only T . Define $Reuse(A_i) := \min\{|A_i(\mathbf{R}_S)|, |A_i(\mathbf{R}_T)|\}$. We then have:

Lemma 4.7. *The I/O cost of a program containing statements S and T that share the input array A_i is bounded by*

$$Q_{tot} \geq Q_S + Q_T - Reuse(A_i),$$

where Q_S, Q_T are the I/O costs of a program containing only statement S or T , respectively.

Proof. Consider an optimal sequential schedule of a CDAG G_S containing statement S only. For any subcomputation H_s and its associated iteration domain \mathbf{R}_s its minimum dominator set is $Dom(H_s) = \bigcup_{j=1}^m A_j(\mathbf{R}_s)$. To compute H_s , at least $\sum_{i=1}^m |A_i(\mathbf{R}_s)| - S$ vertices have to be loaded, as only S vertices can be reused from previous subcomputations.

We seek if any loads can be avoided in the common schedule if we add statement T , denoting its CDAG G_{S+T} . Consider a subset $A_i(\mathbf{R}_x)$ of vertices in A_i .

Consider some subset of vertices in A_i which potentially could be reused and denote it Θ_i . Now denote all vertices in A_0 (statement S) which depend on any vertex from Θ_i as Θ_S , and, analogously, set Θ_T for statement T . Now consider these two subsets Θ_S and Θ_T separately. If Θ_S is computed before Θ_T , then it had to load all vertices from Θ_i , avoiding no loads compared to the schedule of G_S only. Now, computation of Θ_T may take benefit of some vertices from Θ_i , which can still reside in fast memory, avoiding up to $|\Theta_i|$ loads.

The total number of avoided loads is bounded by the number of loads from A_i which are shared by both S and T . Because statement S loads at

most $|A_i(\mathbf{R}_S)|$ vertices from A_i during optimal schedule of G_S , and T loads at most $|A_i(\mathbf{R}_T)|$ of them for G_T , the upper bound of shared, and possibly avoided loads is $Reuse(A_i) = \min\{|A_i(\mathbf{R}_S)|, |A_i(\mathbf{R}_T)|\}$. \square

The **reuse size** is defined as $Reuse(A_i) = \min\{|A_i(\mathbf{R}_S)|, |A_i(\mathbf{R}_T)|\}$. Now, how to find $|A_i(\mathbf{R}_S)|$ and $|A_i(\mathbf{R}_T)|$?

Observe that $|A_i(\mathbf{R}_S)|$ is a property of G_S , that is, the CDAG containing statement S only. Denote the I/O optimal schedule parameters of G_S : V_{max}^S , X_0^S , and $|A_i(\mathbf{R}_{max}^S(X_0^S))|$ (Section 4.3.2). Similarly, for G_T : V_{max}^T , X_0^T , and $|A_i(\mathbf{R}_{max}^T(X_0^T))|$. We now derive: 1) at least how many subcomputations does the optimal schedule have: $s \geq \frac{|V|}{|H_{max}|}$, 2) at least how many accesses to A_i are performed per optimal subcomputation $|A_i(\mathbf{R}_{max}(X_0))|$. Then:

$$Reuse(A_i) = \min\left\{|A_i(\mathbf{R}_{max}^S(X_0^S))| \frac{|V^S|}{|V_{max}^S|}, \quad (4.6)\right. \\ \left.|A_i(\mathbf{R}_{max}^T(X_0^T))| \frac{|V^T|}{|V_{max}^T|}\right\}$$

Note that $Reuse(A_i)$ is an overapproximation of the actual reuse. Since finding the optimal schedule is PSPACE-complete [130], we conservatively assume that only the minimum number of loads from A_i is performed. Thus, Lemma 4.7 generalizes to any number of statements S_1, \dots, S_w sharing array A_i — the total number of loads from A_i is lower-bounded by a maximum number of loads from A_i among S_j , $\max_{j=1, \dots, w} |A_i(\mathbf{R}_{S_j})|$.

4.4.2 Case II: Output Reuse and Access Sizes

Consider the case where the *output* A_0 of statement S is also the *input* B_j of statement T . Furthermore, consider subcomputation H of statement T (and its associated iteration domain D). Any path from the graph inputs to vertices in $B_0(D)$ must pass through vertices in $B_j(D)$. The following question arises: Is there a smaller set of vertices $B'_j(D)$, $|B'_j(D)| < |B_j(D)|$ that every path from graph inputs to $B_j(D)$ must pass through?

Let ρ_S denote computational intensity of statement S . With that, we can state the following lemma:

Lemma 4.8. *Any dominator set of set $B_j(D)$ must be of size at least $|Dom(B_j(D))| \geq \frac{|B_j(D)|}{\rho_S}$.*

Proof. By Lemma 4.1, for one loaded vertex, we may compute at most ρ_S vertices of A_0 . These are also vertices of B_j . Thus, to compute $|B_j(\mathcal{D})|$ vertices of B_j , at least $\frac{|B_j(\mathcal{D})|}{\rho_S}$ loads must be performed. We just need to show that at least that many vertices have to be in any dominator set $Dom(B_j(\mathcal{D}))$. Now, consider the converse: There is a vertex set $D = Dom(B_j(\mathcal{D}))$ such that $|D| < \frac{|B_j(\mathcal{D})|}{\rho_S}$. But that would mean, that we could potentially compute all $|B_j(\mathcal{D})|$ vertices by only loading $|D|$ vertices, violating Lemma 4.1. \square

Corollary 4.1. *Combining Lemmas 4.8 and 4.3, the data access size of $|B_j(\mathcal{D})|$ during subcomputation H is*

$$|Dom(B_j(\mathcal{D}))| \geq \frac{\prod_{k=1}^{dim(\phi_j)} |D_j^k|}{\rho_S}. \quad (4.7)$$

Similarly to **case I**, this result also generalizes to multiple “consumer” statements that reuse the same output array of a “producer” statement, as well as any combination of input and output reuse for multiple arrays and statements. Since the actual I/O optimal schedule is unknown, the general strategy to ensure correctness of our lower bound is to consider each pair of interacting statements separately as one of these two cases. Since both Lemma 4.7 and 4.8 overapproximate the reuse, the final bound may not be tight - the more inter-statement reuse, the more overapproximation is needed. Still, this method can be successfully applied to derive *tight* I/O lower bounds for many linear algebra kernels, such as matrix factorizations, tensor products, or solvers.

4.5 GENERAL PARALLEL I/O LOWER BOUNDS

We now establish how our method applies to a parallel machine with p processors (Section 4.2.1). Since we target large-scale distributed systems, our parallel pebbling model differs from the one introduced e.g. by Alwen and Serbinenko [164], which is inspired by shared-memory models like PRAM [165]. Instead, we disallow sharing memory (pebbles) between the processors, and enforce explicit communication — analogous to the load/store operations — using red and blue pebbles. This allows us to better match the behavior of real, distributed applications that use, e.g., MPI.

Each processor p_i owns its private fast memory that can hold up to S words, represented in the CDAG as S vertices of color p_i . Vertices with different colors (belonging to different processors) cannot be shared between

these processors, but any number of different pebbles may be placed on one vertex.

All the standard red-blue pebble game rules apply with the following modifications:

1. **compute.** If all direct predecessors of vertex v have pebbles of p_i 's color placed on them, one can place a pebble of color p_i on v (no sharing of pebbles between processors),
2. **communication.** If a vertex v has *any* pebble placed on it, a pebble of any other color may be placed on this vertex.

From this game definition it follows that from a perspective of a single processor p_i , any data is either local (the corresponding vertex has p_i 's pebble placed on it) or remote, without a distinction on the remote location (remote access cost is uniform).

Lemma 4.9. *The minimum number of I/O operations in a parallel pebble game, played on a CDAG with $|V|$ vertices with p processors each equipped with S pebbles, is $Q \geq \frac{|V|}{p \cdot \rho}$, where ρ is the maximum computational intensity, which is independent of p (Lemma 4.1).*

Proof. Following the analysis of Section 4.3 and the parallel machine model (Section 4.5), the computational intensity ρ is independent of a number of parallel processors - it is solely a property of a CDAG and private fast memory size S . Therefore, following Lemma 4.1, what changes with p is the volume of computation $|V|$, as now at least one processor will compute at least $|V_p| = \frac{|V|}{p}$ vertices. By the definition of the computational intensity, the minimum number of I/O operations required to pebble these $|V_p|$ vertices is $\frac{|V_p|}{\rho}$. □

4.6 I/O LOWER BOUNDS OF PARALLEL FACTORIZATION ALGORITHMS

We gather all the insights from Sections 4.2 to 4.5 and use them to obtain the parallel I/O lower bounds of LU and Cholesky factorization algorithms, which we use to develop our communication-avoiding implementations.

4.6.1 LU Factorization

In our I/O lower bound analysis we omit the row pivoting, since swapping rows can increase the I/O cost by at most N^2 , which is the cost of permuting

the entire matrix. However, the total I/O cost of the LU factorization is $\mathcal{O}(N^3/\sqrt{5})$ [52].

LU factorization (without pivoting) contains two statements (Figure 4.3). Observe that we can use Lemma 4.6 (out-degree one vertices) for statement $S1 : A[i, k] = A[i, k] / A[k, k]$. The loop nest depth is $l_{S1} = 2$, with iteration variables $\psi^1 = k$ and $\psi^2 = i$. The dimension of the access function vector (k, k) is 1, revealing potential for data reuse: every input vertex $A[k, k]$ is accessed $N - k$ times and used to compute vertices $A[i, k]$, $k + 1 \leq i < N$. However, the access function vector (i, k) has dimension 2; therefore, every compute vertex has one direct predecessor with out-degree one, which is the previous version of element $A[i, k]$. Using Lemma 4.6, we therefore have $\rho_{S1} \leq 1$.

We now proceed to statement $S2 : A[i, j] = A[i, j] - A[i, k] * A[k, j]$. Let $|D^k| = K$, $|D^i| = I$, $|D^j| = J$. Observe that there is an output reuse (Section 4.4.2 and Figure 4.3, red arrow) of $A[i, k]$ between statements $S1$ and $S2$. We therefore have the following access size in statement $S2$: $|A_2(D_{S2})| = |A[i, k]| = \frac{IK}{\rho_{S1}} \geq IK$ (Equation 4.7). Note that in this case where the computational intensity is $\rho_{S1} \leq 1$, the output reuse does not change the access size $|A_2(D_{S2})|$ of statement $S2$. This follows the intuition that it is not beneficial to recompute vertices if the recomputation cost is higher than loading it from the memory. Denoting H_{S2} as the maximal subcomputation for statement $S2$ over the subcomputation domain D , we have the following (Lemma 4.3):

- $|H_{S2}| = KIJ$
- $|A_1(D)| = |A[i, j]| = IJ$
- $|A_2(D)| = |A[i, k]| = IK$
- $|A_3(D)| = |A[k, j]| = KJ$
- $|Dom(H_{S2})| = |A_1(D)| + |A_2(D)| + |A_3(D)| = IJ + IK + KJ$

We then solve the optimization problem from Section 4.3.2:

$$\begin{aligned} \max \quad & KIJ, \quad \text{s.t.} \\ & IJ + IK + KJ \leq X \\ & I \geq 1, \quad J \geq 1, \quad K \geq 1 \end{aligned}$$

Which gives $|H_{S2}| = \chi(X) = \left(\frac{X}{3}\right)^{\frac{3}{2}}$ for $K = I = J = \left(\frac{X}{3}\right)^{\frac{1}{2}}$. Then, we find X_0 that minimizes the expression $\rho_{S2}(X) = \frac{|H_{max}|}{X-5}$ (Equation 4.4),

yielding $X_0 = 3M$. Plugging it into $\rho_{S2}(X)$, we conclude that the maximum computational intensity of $S2$ is bounded by $\rho_{S2} \leq \sqrt{5}/2$.

We bounded the maximum computational intensities ρ_{S1} and ρ_{S2} , that is, the minimum number of I/O operations to compute vertices belonging to statements $S1$ and $S2$. As the last step, we find the total number of compute vertices for each statement: $|V_1| = \sum_{k=1}^N (N - k - 1) = N(N - 1)/2$, and $|V_2| = \sum_{k=1}^N \sum_{i=k+1}^N (N - k - 1) = N(N - 1)(N - 2)/3$. Using Lemmas 4.1 (bounding I/O cost with the computational intensity) and 4.9 (I/O cost of the parallel machine), the parallel I/O lower bound for LU factorization is therefore

$$Q_{p,LU} \geq \frac{2N^3 - 6N^2 + 4N}{3P\sqrt{S}} + \frac{N(N - 1)}{2P} = \frac{2N^3}{3P\sqrt{S}} + \mathcal{O}\left(\frac{N^2}{p}\right).$$

Previously, Solomonik et al. [52] established the asymptotic I/O bound for sequential execution $Q = \mathcal{O}(N^3/\sqrt{S})$. Recently, Olivry et al. [162] derived a tight leading term constant $Q \geq 2N^3/(3\sqrt{S})$. To the best of our knowledge, our result is the first non-asymptotic bound for parallel execution. The generalization from the sequential to the parallel bound is straightforward. Note, however, that this is only the case due to our pebble-based execution model, and it may thus not apply to other parallel machine models.

4.6.2 Cholesky Factorization

We proceed analogously to our derivation of the LU I/O bound — here we just briefly outline the steps. The algorithm contains three statements (Listing 4.1). For statements $S1$ and $S2$, we can again use Lemma 4.6 (out-degree-one vertices). For $S1 : L(k, k) = \text{sqrt}(L(k, k))$, the loop nest depth is $l_1 = 1$, we have a single iteration variable $\psi^1 = k$, and a single input array $A_1 = L$ with the access function $\phi_1(\psi) = (k, k)$. Since there is only one iteration variable present in ϕ_1 , we have $\dim(\phi_1) = 1 = l_1$. Therefore, for every compute vertex v we have one direct predecessor, which is the previous version of element $L(k, k)$. We conclude that $\rho_{S1} \leq 1$ and $|V_{S1}| = N$.

For statement $S2 : L(i, k) = (L(i, k)) / L(k, k)$, we also have output reuse of $L(k, k)$ between statements $S2$ and $S1$. However, as with the output reuse considered in the LU analysis, the computational intensity is $\rho_{S1} \leq 1$. Therefore, it does not change the dominator set size of $S2$. We then use the same reasoning as for the corresponding statement $S1$ in LU factorization, yielding $\rho_{S2} \leq 1$.

Listing 4.1 Cholesky Factorization

```

1   for k = 1:N
2   S1:   L(k,k) = sqrt(L(k,k));
3       for i = k+1:N
4   S2:   L(i,k) = (L(i,k)) / L(k,k);
5       for j = k+1:i
6   S3:   L(i,j) = L(i,j) - L(i,k) * L(j,k);
7   end; end; end;

```

For statement S3, we derive its bound similarly to S2 of LU, with $\rho_{S3} = \sqrt{S}/2$ and $|V_{S3}| = \sum_{k=1}^N \sum_{i=k+1}^N (i-k-1) = N(N-1)(N-2)/6$. Note that compared to LU, the only significant difference is the iteration domain $|V_3|$. Even though Cholesky has one statement more – the diagonal element update $L(k,k)$ – its impact on the final I/O bound is negligible for large N .

Again, using Lemmas 4.1 and 4.9 we establish the Cholesky factorization’s parallel I/O lower bound:

$$Q_{Chol} \geq Q_1 + Q_2 + Q_3 = \frac{|V_1|}{p\rho_1} + \frac{|V_2|}{p\rho_2} + \frac{|V_3|}{p\rho_3} \approx \frac{N^3}{3p\sqrt{S}} + \frac{N^2}{2P} + \frac{N}{p}$$

The derived I/O lower bound for a sequential machine ($p = 1$) improves the previous bound $Q_{chol} \geq N^3/(6\sqrt{S})$ derived by Olivry et al. [162]. Furthermore, to the best of our knowledge, this is the first parallel bound for this kernel.

4.7 NEAR-I/O OPTIMAL PARALLEL MATRIX FACTORIZATION ALGORITHMS

We now present our parallel LU and Cholesky factorization algorithms. We start with the former, more complex algorithm, i.e. LU factorization. Pivoting in LU poses several performance challenges. First, since pivots are not known upfront, additional communication and synchronization is required to determine them in each step. Second, the nondeterministic pivot distribution between the ranks may introduce load imbalance of computation routines. Third, to minimize the communication a 2.5D parallel decomposition must be used, i.e. parallelization along the reduction dimension. We address all these challenges with *CONfLUX* — a near *Communication Optimal LU factorization using X-Partitioning*.

4.7.1 LU Dependencies and Parallelization

Due to the dependency structure of LU, the input matrix is often divided recursively into four submatrices A_{00} , A_{10} , A_{01} , and A_{11} [52, 166]. Arithmetic operations performed in LU create non-commutative dependencies (Figure 4.6) between vertices in A_{00} (LU factorization of the top-left corner of the matrix), A_{10} , and A_{01} (triangular solve of left and top panels of the matrix). Only A_{11} (Schur complement update) has no such dependencies, and may therefore be efficiently parallelized in the reduction dimension. A high-level summary is presented in Algorithm 2.

Algorithm 2 ConfLUX

Input: Input matrix $A \in \mathbb{R}^{N \times N}$

Output: In-place factored matrix A , permutation matrix P

$A_1 \leftarrow A$ ▷ First step
 $P \leftarrow I$ ▷ Permutation matrix is initially identity
for $t = 1, \dots, \frac{N}{v}$ **do**
 1. Reduce next block column ▷ Cost: $\frac{(N-t)v \cdot v \cdot S}{N^2}$
 2. $[rows, P_{t+1}] \leftarrow \text{TournPivot}(A_t, P_t)$ ▷ Find next v pivots. Cost: $v^2 \left[\log\left(\frac{N}{\sqrt{S}}\right) \right]$
 3. Scatter computed A_{00} and v pivot rows ▷ Cost: $v^2 + v$
 4. Scatter A_{10} ▷ Cost: $\frac{(N-t)v}{p}$
 5. Reduce v pivot rows ▷ Cost: $\frac{(N-t)v \cdot v \cdot S}{N^2}$
 6. Scatter A_{01} ▷ Cost: $\frac{(N-t)v}{p}$
 7. $\text{Factorize}A_{10}(A_t)$ ▷ 1D parallel., block-row
 8. Send data from panel A_{10} ▷ Cost: $\frac{(N-t)vN \cdot v}{p\sqrt{S}}$
 9. $\text{Factorize}A_{01}(A_t)$ ▷ 1D parallel., block-column
 10. Send data from panel A_{01} ▷ Cost: $\frac{(N-t)vN \cdot v}{p\sqrt{S}}$
 11. $\text{Factorize}A_{11}(A_t)$ ▷ 2.5D parallel.
 $A_{t+1} \leftarrow A_t[rows, v : end]$ ▷ Recursively process remaining rows and columns
end for

4.7.2 LU Computation Routines

The computation is performed in $\frac{N}{v}$ steps, where v is a tunable block size. In each step, only submatrix A_t of input matrix A is updated. Initially, A_t is set to A . A_t can be further viewed as composed of four submatrices A_{00} , A_{10} , A_{01} , and A_{11} (see Figure 4.7). These submatrices are dis-

tributed and updated by routines *TournPivot*, *FactorizeA₁₀*, *FactorizeA₀₁*, and *FactorizeA₁₁*:

- **A₀₀**. This $v \times v$ submatrix contains the first v elements of the current v pivot rows. It is computed during *TournPivot*, and, as it is required to compute A_{10} and A_{01} , it is redundantly copied to all processors.
- **A₁₀ and A₀₁**. Submatrices A_{10} and A_{01} of sizes $(N - t \cdot v) \times v$ and $v \times (N - t \cdot v)$ are distributed using a 1D decomposition among all processors. They are updated using a triangular solve. 1D decomposition guarantees that there are no dependencies between processors, so no communication or synchronization is performed during computation, as A_{00} is already owned by every processor.
- **A₁₁** This $(N - t \cdot v) \times (N - t \cdot v)$ submatrix is distributed using a 2.5D, block-cyclic distribution (Figure 4.7). First, the updated submatrices A_{10} and A_{01} are broadcast among the processors. Then, A_{11} (Schur complement) is updated. Finally, the first block column and v chosen pivot rows are reduced, which will form A_{10} and A_{01} in the next iteration.

Block size v . The minimum size of each block is the number of processor layers in the reduction dimension ($v \geq c = \frac{pS}{N^2}$). However, to secure high performance, this value should be adjusted to hardware parameters of a given machine (e.g., vector length, prefetch distance of a CPU, or warp size of a GPU). Throughout the analysis, we assume $v = a \cdot \frac{pS}{N^2}$ for some small constant a .

4.7.3 Pivoting

Our pivoting strategy differs from state-of-the-art block [167], tile [168], or recursive [166] pivoting approaches in two aspects:

- To minimize I/O cost, we do not swap pivot rows. Instead, we keep track of which rows were chosen as pivots and we use masks to update the remaining rows.
- To reduce latency, we take advantage of our derived block decomposition and use tournament pivoting [169].

Tournament Pivoting. This procedure finds v pivot rows in each step that are then used to mask which rows will form the new A_{01} and then filter the non-processed rows in the next step. It is shown to be as stable as partial

		CO _{nf} LUX (LU)			CO _{nf} CHOX (Cholesky)		
		description	comm. cost	comp. cost	description	comm. cost	comp. cost
pivoting	<i>TournPivot</i>	$v^2 \lceil \log_2(\sqrt{p1}) \rceil$	$v^3 / 3 \lceil \log_2(\sqrt{p1}) \rceil$		(no pivoting)	—	—
A_{00}	local getrf	o	o	(done during <i>TournPivot</i>)	potrf	v^2	$v^3 / 6$
A_{10} and A_{01}	reduction, local trsm	$\frac{2(N-tv)vM}{N^2}$	$\frac{2(N-tv)v^2}{2P}$		(similar to LU)	$\frac{2(N-tv)vM}{N^2}$	$\frac{2(N-tv)v^2}{2P}$
A_{11}	scatter, local gemm	$\frac{2(N-tv)v}{p}$	$\frac{(N-tv)^2v}{p}$		scatter, local gemmt (triangular gemm)	$\frac{2(N-tv)v}{p}$	$\frac{(N-tv)^2v}{2P}$

Table 4.1: Comparison of the implemented LU and Cholesky factorizations. Even though Cholesky performs half as many computations (the use of `gemmt` instead of `gemm` in A_{11}), it communicates the same amount of data, since the number of elements needed to perform `gemm` and `gemmt` is the same.

pivoting [169], which might be an issue for, e.g., incremental pivoting [170]. On the other hand, it reduces the $\mathcal{O}(N)$ latency cost of partial pivoting, which requires step-by-step column reduction to find consecutive pivots, to $\mathcal{O}(\frac{N}{v})$, where v is the tunable block size parameter.

Row Swapping vs. Row Masking. To achieve close to optimal I/O cost, we use a 2.5D decomposition. This, however, implies that in the presence of extra memory, the matrix data is replicated $\frac{pS}{N^2}$ times. This increases the row swapping cost from $\mathcal{O}(\frac{N^2}{p})$ to $\mathcal{O}(\frac{N^3}{p\sqrt{S}})$, which asymptotically matches the I/O lower bound of the entire factorization. Performing row swapping would then increase the constant term of the leading factor of the algorithm from $\frac{N^3}{p\sqrt{S}}$ to $\frac{2N^3}{p\sqrt{S}}$. To keep the I/O cost of our algorithm as low as possible, instead of performing row-swapping, we only propagate pivot row indices. When the tournament pivoting finds the v pivot rows, they are broadcast to all processors with only v cost per step.

Pivoting in ConfLUX. In each step t of the outer loop (line 1 in Algorithm 2), $\frac{N}{\sqrt{S}}$ processors perform a tournament pivoting routine using a butterfly communication pattern [171]. Each processor owns $\sqrt{S}\frac{N-vt}{N}$ rows, among which it chooses v local candidate pivots. Then, final pivots are chosen in $\log(\frac{N}{\sqrt{S}})$ “playoff-like” tournament rounds, after which all $\frac{N}{\sqrt{S}}$ processors own both v pivot row indices and the already factored, new A_{00} . This result is distributed to all remaining processors (line 2). Pivot row indices are then used to determine which processors participate in the reduction of the current A_{01} (line 4). Then, the new A_t is formed by masking currently chosen rows $A_t \leftarrow A_t[\text{rows}, v : \text{end}]$ (Line 12).

4.7.4 I/O cost of ConfLUX

We now prove the I/O cost of ConfLUX, which is only a factor of $\frac{1}{3}$ higher than the obtained lower bound for large N .

Lemma 4.10. *The total I/O cost of ConfLUX, presented in Algorithm 2, is $Q_{\text{ConfLUX}} = \frac{N^3}{p\sqrt{S}} + \mathcal{O}\left(\frac{N^2}{p}\right)$.*

Proof. We assume that the input matrix A is already distributed in the block cyclic layout imposed by the algorithm. Otherwise, data reshuffling imposes only $\Omega\left(\frac{N^2}{p}\right)$ cost, which does not contribute to the leading order term. We first derive the cost of a single iteration t of the main loop of the algorithm,

proving that its cost is $Q_{step}(t) = \frac{2Nv(N-tv)}{p\sqrt{S}} + \mathcal{O}\left(\frac{Nv}{p}\right)$. The total cost after $\frac{N}{v}$ iterations is:

$$Q_{ConflUX} = \sum_{t=1}^{\frac{N}{v}} Q_{step}(t) = \frac{N^3}{p\sqrt{S}} + \mathcal{O}\left(\frac{N^2}{p}\right).$$

We define $p1 = \frac{N^2}{S}$ and $c = \frac{pS}{N^2}$. p processors are decomposed into the 3D grid $[\sqrt{p1}, \sqrt{p1}, c]$. We refer to all processors that share the same second and third coordinate as $[:, j, k]$. We now examine each of 11 steps in Algorithm 2.

Step 2. Processors with coordinates $[:, t \bmod \sqrt{p1}, t \bmod c]$ perform the tournament pivoting. Every processor owns the first v elements of $N - (t - 1)v$ rows, among which they choose the next v pivots. First, they locally perform the LUP decomposition to choose the local v candidate rows. Then, in $\lceil \log_2(\sqrt{p1}) \rceil$ rounds they exchange $v \times v$ blocks to decide on the final pivots. After the exchange, these processors also hold the factorized submatrix A_{00} . *I/O cost per processor:* $v^2 \lceil \log_2(\sqrt{p1}) \rceil$.

Steps 3, 4, 6. Factorized A_{00} and v pivot row indices are broadcast. First v columns and v pivot rows are scattered to all p . *I/O cost per processor:* $v^2 + v + \frac{2(N-tv)v}{p}$.

Steps 1 and 5. v columns and v pivot rows are reduced. With high probability, pivots are evenly distributed among all processors. There are c layers to reduce, each of size $(N - tv)v$. *I/O cost per processor:* $\frac{(N-tv)vc}{p} = \frac{2(N-tv)vM}{N^2}$.

Steps 7, 9, 11. The updates $FactorizeA_{10}$, $FactorizeA_{01}$, and $FactorizeA_{11}$ are local and incur no additional I/O cost.

Steps 8 and 10. Factorized A_{10} and A_{01} are scattered among all processors. Each processor requires $\frac{v(N-tv)}{c\sqrt{p1}}$ elements from A_{10} and A_{01} . *I/O cost per processor:* $\frac{2(N-tv)Nv}{p\sqrt{S}}$.

Summing steps 1 – 11: $Q_{step}(t) = \frac{2Nv(N-tv)}{p\sqrt{S}} + \mathcal{O}\left(\frac{Nv}{p}\right)$. □

Note that this cost is a factor $1/3$ over the lower bound established in Section 4.6.1. This is due to the fact that any processor can only maximally utilize its local memory in the first iteration of the outer loop. In this first iteration, a processor updates a total of $\sqrt{S} \times \sqrt{S}$ elements of A . In subsequent iterations, however, the local domain shrinks as less rows and columns are updated, which leads to an underutilization of the resources. Since the shape of the iteration space is determined by the algorithm, this

behavior is unavoidable for $p \geq N^2/S$. Note that the bound is attainable by a sequential machine, however.

4.7.5 Cholesky Factorization

From a data flow perspective, Cholesky factorization can be viewed as a special case of LU factorization without pivoting for symmetric, positive definite matrices. Therefore, our Cholesky algorithm — *ConfCHOX* — heavily bases on *ConfLUX*, using the same 2.5D parallel decomposition, block-cyclic data distribution, and analogous computation routines.

For both algorithms, the dominant cost, both in terms of computation and communication, is the A_{11} update. Due to the Cholesky factorization’s iteration domain, which exploits the symmetry of the input matrix, the compute cost is twice as low, as only one half of the matrix needs to be updated. However, the input size required to perform this update is the same — therefore, the communication cost imposed by A_{11} is similar. We list the key differences between the two factorization algorithms in Table 4.1.

4.8 IMPLEMENTATION

Our algorithms are implemented in C++, using MPI for inter-node communication. For static communication patterns (e.g., column reductions) we use dedicated, asynchronous MPI collectives. For runtime-dependent communication (e.g., pivot index distribution) we use MPI one-sided [133]. For intra-node tasks, we use OpenMP and local BLAS calls (provided by Intel MKL [158]) for computations. Our code is available as an open-source git repository¹.

Parallel decomposition. Our experiments show that the parallelization in the reduction dimension, while reducing communication volume, does incur performance overheads. This is mainly due to the increased communication latency, as well as smaller buffer sizes used for local BLAS calls. Since formal modeling of the tradeoff between communication volume and performance is outside of the scope of the paper, we keep the depth of parallelization in the third dimension as a tunable parameter, while providing heuristics-based default values.

Data distribution. *ConfLUX* and *ConfCHOX* provide ScaLAPACK wrappers by using the highly-optimized COSTA algorithm [172] to transform the matrices between different layouts. In addition, they support the COSTA

¹ <https://github.com/eth-cscs/conflux>













	MKL [158]	SLATE [156]	CANDMC [157]	CAPITAL [155]	CONfLUX / CONfCHOX
Decomposition	2D, panel decomp.	2D, block decomp.	Nested 2.5D, block decomp.	2.5D, block decomp.	1D / 2.5D, block decomp.
Block size	 user- specified	 user- specified, (default 16)	 $\frac{N^3}{p \cdot S}$	$\frac{N^2}{p\sqrt{S}}$  user- specified	 optimized, $\geq \frac{p \cdot S}{N^2}$
Program parameters	required from user 	required from user 	provided defaults 	optimized defaults  	optimized defaults  
Parallel I/O cost	$\frac{N^2}{\sqrt{p}} + \mathcal{O}\left(\frac{N^2}{p}\right)$	$\frac{N^2}{\sqrt{p}} + \mathcal{O}\left(\frac{N^2}{p}\right)$	$\frac{5N^3}{p\sqrt{S}}$ $\mathcal{O}\left(\frac{N^2}{p\sqrt{S}}\right)$	$+$ $\frac{45N^3}{8p\sqrt{S}}$ $\mathcal{O}\left(\frac{N^2}{p\sqrt{S}}\right)$	$+$ $\frac{N^3}{p\sqrt{S}} + \mathcal{O}\left(\frac{N^2}{p\sqrt{S}}\right)$

Table 4.2: Parallelization strategies and I/O cost models of the considered matrix factorization implementations. MKL and SLATE require a user to specify the processor decomposition and the block size. CANDMC provides default values, but our experiments show that the performance was significantly improved when we tuned the parameters. CONfLUX and CONfCHOX outperform all state-of-the-art libraries with out-of-the-box parameters. We validated our parallel I/O cost models: for MKL, SLATE, CONfLUX, and CONfCHOX, the error was within +/-3%. For CANDMC and CAPITAL, we used the models derived by the authors [52, 155], which overapproximated the measured values by approx. 30-40%.

API for matrix descriptors, which is more general than ScaLAPACK’s layout, as it supports matrices distributed in arbitrary grid-like layouts, processor assignments, and local blocks orderings.

4.9 EXPERIMENTAL EVALUATION

We compare CONfLUX and CONfCHOX with state-of-the-art implementations of corresponding distributed matrix factorizations.

Measured values. We measure both the I/O cost and total time-to-solution. For I/O, the aggregate communication volume in distributed runs is counted using the Score-P profiler [173]. We provide both measured values and theoretical cost models. Local `std::chrono` calls are used for time measurements and the maximum execution time among all ranks is reported.

Infrastructure and Measurement. We run our experiments on the XC40 partition of the CSCS Piz Daint supercomputer which comprises 1,813 CPU nodes equipped with Intel Xeon E5-2695 v4 processors (2x18 cores, 64 GiB

DDR3 RAM), interconnected by the Cray Aries network with a Dragonfly network topology. Since the CPUs are dual-socket, two MPI ranks are allocated per compute node.

Comparison Targets. We use 1) Intel MKL (v19.1.1.217). While the library is proprietary, our measurements reaffirm that, like ScaLAPACK, the implementation uses the suboptimal 2D processor decomposition; 2) SLATE [156] — a state-of-the-art distributed linear algebra framework targeted at exascale supercomputers; 3) the latest version of the CANDMC and CAPITAL libraries [159, 160], which use an asymptotically-optimal 2.5D decomposition. The implementations and their characteristics are listed in Table 4.2.

Problem Sizes. We evaluate the algorithms starting from 2 compute nodes (4 MPI ranks) up to 512 nodes (1,024 ranks). For each node count, matrix sizes range from $N = 2,048$ to $N = 2^{19} = 524,288$, provided they fit into the allocated memory (e.g., LU or Cholesky factorization on a double-precision input matrix of dimension $262,144 \times 262,144$ cannot be run on less than 32 nodes). Runs in which none of the libraries achieved more than 3% of the hardware peak are discarded since by adding more nodes the performance starts to deteriorate.

Our benchmarks reflect real-world problems in scientific computing. The High-Performance Linpack benchmark uses a maximal size of $N = 16,473,600$ [174]. In quantum physics, matrix size scales with 2^{qubits} . In physical chemistry or density functional theory (DFT), simulations require factorizing matrices of atom interactions, yielding sizes ranging from $N = 1,024$ up to $N = 131,072$ [40, 175]. In machine learning, matrix factorizations are used for inverting Kronecker factors [176] whose sizes are usually around $N = 4,096$. This motivates us to focus not only on exascale problems, but also improve performance for relatively small matrices ($N \leq 100,000$).

Communication Models. Together with empirical measurements, we put significant effort into understanding the underlying communication patterns of the compared LU factorization implementations. Both MKL and SLATE base on the standard partial pivoting algorithm using the 2D decomposition [119]. For CANDMC and CAPITAL, the models provided by the authors [52, 155] are used. For *CONfLUX* and *CONfCHOX*, we use the results from Section 4.7. These models are summarized in Table 4.2.

4.10 RESULTS

Our experiments confirm advantages of *CONfLUX* and *CONfCHOX* in terms of both communication volume and time-to-solution over all other implementations tested. A significant communication reduction can be observed (up to 1.42 times for *CONfLUX* compared with the second-best implementation for $p = 1,024$). Moreover, the performance models predict even greater benefits for larger runs (expected 2.1 times communication reduction for a full-machine run on the Summit supercomputer – Figure 4.8c). Most importantly, our implementations consistently outperform existing implementations (up to three times – Figures 4.1 and 4.9).

Communication volume. Fig. 4.8a presents the measured communication volume per node, as well as our derived cost models (Table 4.2) presented with solid lines, for $N = 16,384$. Observe that *CONfLUX* communicates the least for all values of p . Note that since both MKL and SLATE use similar 2D decompositions, their communication volumes are mostly equal, with a slight advantage for SLATE. In Fig. 4.8b, we show the weak scaling characteristics of the analyzed implementations. Observe that for a fixed amount of work per node, the 2D algorithms - MKL and SLATE - scale sub-optimally. Figure 4.8c summarizes the communication volume reduction of *CONfLUX* compared with the second-best implementation, both for measurements and theoretical predictions. It can be seen that for all combinations of p and N , *CONfLUX* always communicates the least. For all measured data points, the asymptotically optimal CANDMC performed worse than MKL or SLATE. The figure also presents the predicted communication cost of all considered implementations for up to $p = 262,144$ based on our theoretical models.

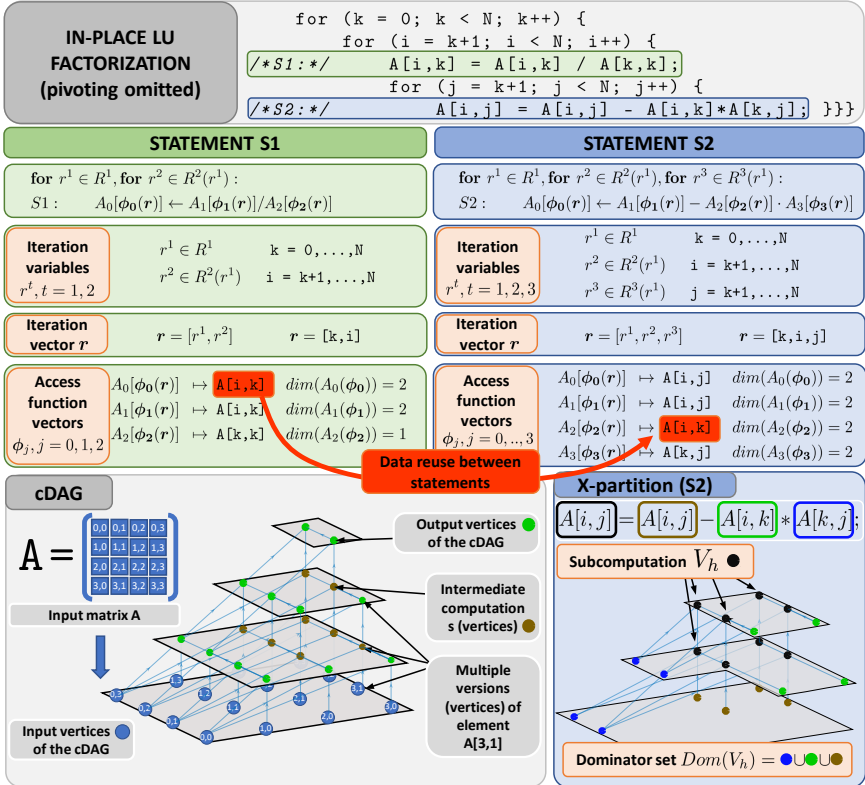


Figure 4.3: In-place LU factorization (for simplicity, no pivoting is performed). The algorithm contains two statements (S1 and S2), for which we provide key components of our program representation together with the corresponding CDAG for $N = 4$. For statement S2, we also provide a graphical visualization of a single subcomputation \mathcal{H} in its X-partition.

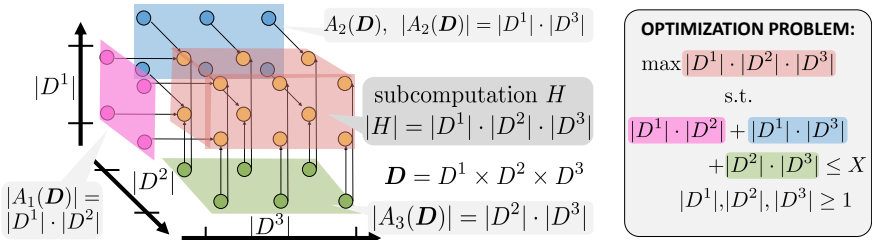


Figure 4.4: Lemma 4.3 bounds the set sizes (both the subcomputation's H and input access sets' $|A_j(D)|$) with the number of values $|D^t|$ each iteration variable ψ^t takes during the subcomputation.

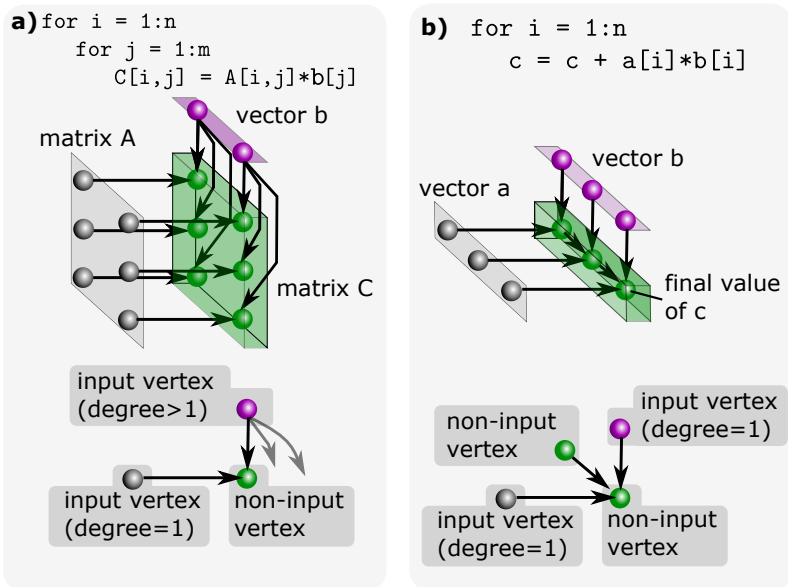


Figure 4.5: CDAGs with out-degree 1 input vertices. a) $u_a = 1$, $\rho_a \leq 1$. b) $u_b = 2$, $\rho_b \leq \frac{1}{2}$.

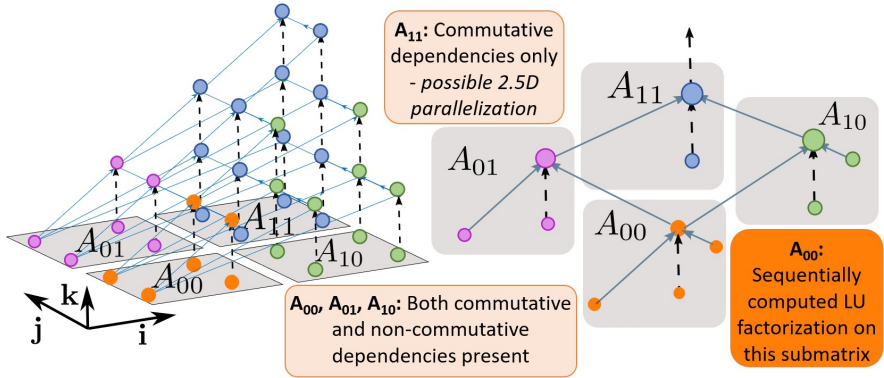


Figure 4.6: LU Factorization CDAG for $N = 4$ with the logical decomposition into A_{00}, A_{10}, A_{01} , and A_{11} . Dashed arrows represent commutative dependencies (reduction of a value). Solid arrows represent non-commutative operations, meaning that any parallel pebbling has to respect the induced order (e.g., no vertex in A_{11} can be pebbled before A_{00} is pebbled).

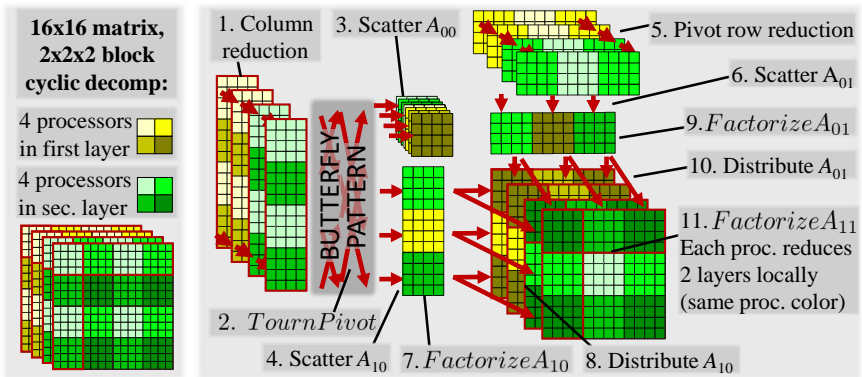
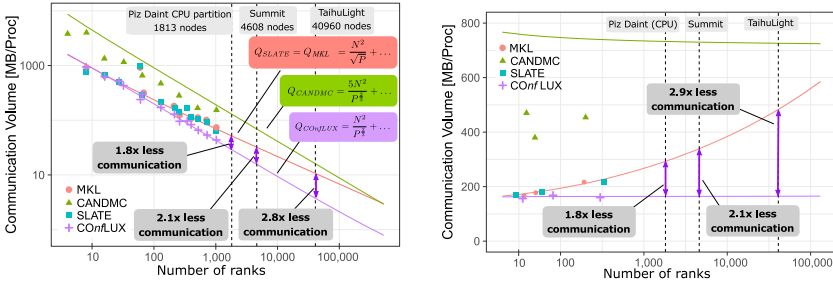
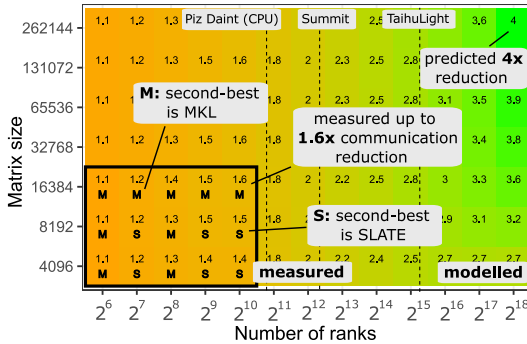


Figure 4.7: CONfLUX's parallel decomposition for $p = 8$ processors decomposed into a $[P_x, P_y, P_z] = [2, 2, 2]$ grid, together with the indicated steps of Algorithm 2. In each iteration t , each processor $[p_i, p_j, p_k]$ updates $(2 - \lfloor (t + p_i) / P_x \rfloor) \times (2 - \lfloor (t + p_j) / P_y \rfloor)$ tiles of A_{11} . In the presented example, there are $v = 4$ planes in dimension k to be reduced, which are distributed among $P_z = 2$ processor layers (green and yellow tiles).

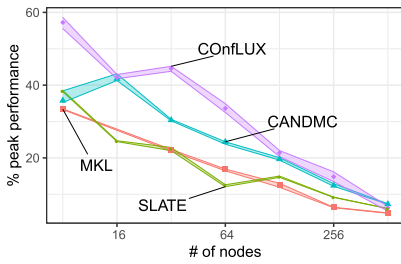


(a) Communication volume per node for varying node counts p and a fixed $N = 16,384$. Only the leading factors of the models are shown. The models are scaled by the element size (8 bytes).
 (b) Communication volume per node for weak scaling (constant work per node), $N = 3200 \cdot \sqrt[3]{p}$. 2.5D algorithms (CANDMC and ConfLUX) retain constant communication volume per processor.

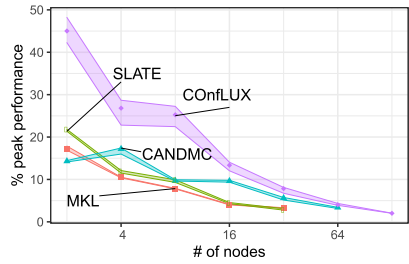


(c) Communication reduction vs. second-best algorithm (S=MKL, S=SLATE), for varying p, N , for both measured and predicted scenarios.

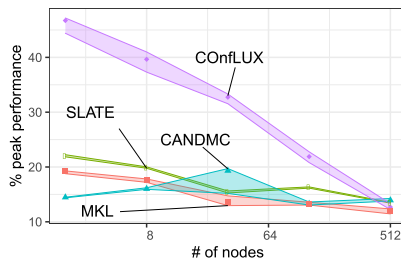
Figure 4.8: Communication volume measurements across different scenarios for MKL, SLATE, CANDMC, and ConfLUX. In all considered scenarios, enough memory $S \geq N^2/p^{2/3}$ was present to allow for the maximum number of replications $c = p^{1/3}$.



(a) Strong scaling, $N = 2^{17} = 131,072$



(b) Strong scaling, $N = 2^{14} = 16,384$



(c) Weak scaling, $N = 8,192 \cdot \sqrt{p}$

Figure 4.9: Achieved % of peak performance for LU factorization. We show median and 95% confidence intervals.

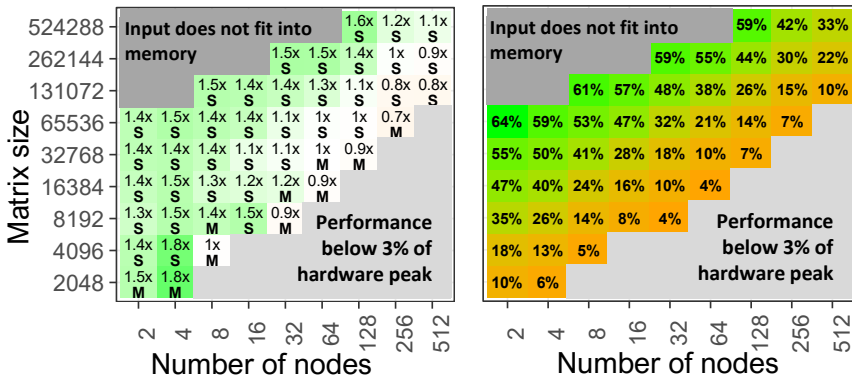


Figure 4.10: Left: measured runtime speedup of COmfCHOX vs. fastest state-of-the-art library (S=SLATE [156], C=CAPITAL [155], S=MKL [158]). Right: COmfCHOX's achieved % of machine peak performance.

Performance. Our measurements show that both *CONfLUX* and *CONfCHOX* outperform all considered state-of-the-art libraries in almost all scenarios (Figures 4.1 and 4.10). Thanks to the optimized block data decomposition and efficient overlap of computation and communication, our implementations achieve high performance already on relatively small matrices (approx. 40% of hardware peak for cases where $N^2/p > 2^{27}$). In cases where the local domain per processor becomes very small ($N^2/p < 2^{27}$) our block decomposition does not add that much benefit, since the performance is mostly latency-bound, and not bandwidth-bound. This is visible not only in strong scaling (Figures 4.9 and 4.11, **a**) and **b**), but also in weak scaling (**c**), where the input size per processor N^2/p is constant. This is again caused by latency overheads of scattering data between 1D and 2.5D layouts.

However, as the local domains become larger and may be more efficiently pipelined and overlapped using asynchronous MPI routines and intra-node OpenMP parallelism, the advantage becomes significant (Figures 4.9 and 4.11). *CONfLUX* outperforms existing libraries up to three times (for $p = 4, N = 4096$, second-best library is SLATE – Figure 4.1) and *CONfCHOX* achieves up to 1.8 times speedup (e.g., $p = 4, N = 4,096$, second-best is again SLATE).

Implications for Exascale. Both the communication models' predictions (Figure 4.8c) and measured speedups (Figures 4.1 and 4.10) allow us to predict that when running our implementations on exascale machines, we can expect to see further performance improvements over state-of-the-art libraries. Furthermore, throughput-oriented hardware, such as GPUs and FPGAs, may benefit even more from the communication reduction of our schedules. Thus, *CONfLUX* and *CONfCHOX* not only outperform the state-of-the-art libraries at relatively small scales — which are most common use cases in practice [40, 175, 176] — but also promise speedups on full-scale performance runs on modern supercomputers.

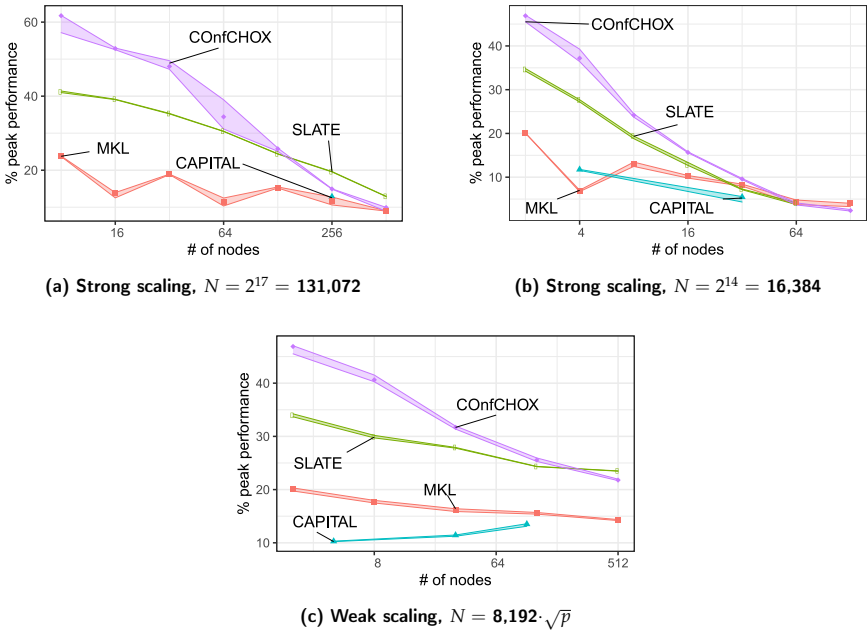


Figure 4.11: Achieved % of peak performance for Cholesky factorization. We show median and 95% confidence intervals.

4.11 RELATED WORK

Previous work on I/O analysis can be categorized into three classes (see Table 4.3): work based on **direct pebbling** or variants of it, such as Vitter’s block-based model [178]; works using **geometric arguments of projections** based on the Loomis-Whitney inequality [131]; and works applying optimizations limited to specific structural properties such as **affine loops** [179], and more generally, **the polyhedral model program representation** [43, 50, 162]. Although the scopes of those approaches significantly overlap — for example, kernels like matrix multiplication can be captured by most of the models — there are important differences both in methodology and the end-results they provide, as summarized in Table 4.3.

Dense linear algebra operators are among the standard core kernels in scientific applications. Ballard et al. [55] present a comprehensive overview

	Pebbling [22, 24, 65, 128, 177]	Projection-based [49, 55, 57–59, 162]	Problem specific [23, 43, 50, 138, 175]
Scope	👍👍 General cDAGs	👍 Programs with a geometric structure of the iteration space	👎 Individually tailored for a given problem
Key Features	👍 General scope 👍 Expresses complex data dependencies 👍 Directly exposes schedules 👍 Intuitive 👎 PSPACE-complete in general case 👎 No guarantees that a solution exists 👎 No well-established method how to automatically translate code to cDAGs	👍 Well-developed theory and tools 👍 Guaranteed to find solution for given class of programs 👎 Bounds are often not tight 👎 Fails to capture dependencies between statements 👎 Limited scope	👍 Takes advantage of problem-specific features 👍 Tends to provide best results in practice 👎 Requires large manual effort for each algorithm separately 👎 Difficult to generalize 👎 Often based on heuristics with no guarantees on optimality

Table 4.3: Overview of different approaches to modeling data movement.

of their asymptotic I/O lower bounds and I/O minimizing schedules, both for sparse and dense matrices. Recently, Olivry et al. introduced IOLB [162] — a framework for assessing sequential lower bounds for polyhedral programs. However, their computational model disallows recomputation (cf. Section 4.4.2).

Matrix factorizations are included in most of linear solvers’ libraries. With regard to the parallelization strategy, these libraries may be categorized into three groups: **task-based**: SLATE [156] (OpenMP tasks), DLAF [180] (HPX tasks), DPLASMA [181] (DaGuE scheduler), or CHAMELEON [182] (StarPU tasks); **static 2D parallel**: MKL [158], Elemental [183], or Cray LibSci [184]; **communication-minimizing 2.5D parallel**: CANDMC [157] and CAPITAL [155]. In the last decade, heavy focus was placed on heterogeneous architectures. Most GPU vendors offer hardware-customized BLAS solvers [185]. Agullo et. al [186] accelerated LU factorization using up to 4 GPUs. Azzam et. al [47] utilize NVIDIA’s GPU tensor cores to compute low-precision LU factorization and then iteratively refine the linear problem’s

solution. Moreover, some of the distributed memory libraries support GPU offloading for local computations [156].

4.12 SUMMARY

In this chapter, we present a method of analyzing I/O cost of DAAP — a general class of programs that covers many fundamental computational motifs. We show, both theoretically and in practice, that our pebbling-based approach for deriving the I/O lower bounds is **more general**: programs with disjoint array accesses cover a wide variety of applications, **more powerful**: it can explicitly capture inter-statement dependencies, **more precise**: it derives tighter I/O bounds, and **more constructive**: X-partition provides powerful hints for obtaining parallel schedules.

When applying the approach to LU and Cholesky factorizations, we are able to derive new lower bounds, as well as new, communication-avoiding schedules. Not only do they communicate less than state-of-the-art 2D *and* 3D decompositions — by a factor of up to $1.6\times$ — but most importantly, they outperform existing commercial libraries in a wide range of problem parameters (up to $3\times$ for LU, up to $1.8\times$ for Cholesky). Finally, our code is openly available, offering full ScaLAPACK layout compatibility.

PRECISE DATA MOVEMENT MODELLING FOR GENERAL CLASS OF PROGRAMS

This is the work published at SPAA'21 conference [25]. It further extends the DAAP class of programs from the previous chapter to capture stencil programs. Furthermore, it precisely formalizes data reuse between statements with the new abstraction - SDG. I want to appreciate help from Tal Ben-Nun in implementing the SOAP analyzer in Python using the DaCe framework.

5.1 INTRODUCTION

I/O operations, both across the memory hierarchy and between parallel processors, dominate time and energy costs in many scientific applications [153, 154, 187]. It is thus of key importance to design algorithms with communication-avoiding or *I/O-efficient* schedules [37, 113]. To inform, and occasionally inspire the development of such algorithms, one must first understand *the associated lower bounds on the amounts of communicated data*. Deriving these bounds has always been of theoretical interest [22, 57]. It is particularly relevant for dense linear algebra, as many important problems in scientific computing [40, 121] and machine learning [188] rely on linear algebra operations such as matrix factorization [104, 150] or tensor contractions [189].

Analyzing I/O bounds of linear algebra kernels dates back to the seminal work by Hong and Kung [22], who derived the first asymptotic bound for matrix-matrix multiplication (MMM) using the red-blue pebble game abstraction. This method was subsequently extended and used by other works to derive asymptotic [190] and tight [65] bounds for more complex programs. Despite the expressiveness of pebbling, it is prohibitively hard to solve for arbitrary programs, as it is PSPACE-complete in the general case [191].

Since analyzing programs with parametric sizes disallows the construction of an explicit Computation Directed Acyclic Graph (CDAG), some form of parameterization is often needed [24, 162, 192]. However, we argue that the widely-used approaches based on the Loomis-Whitney or the HBL

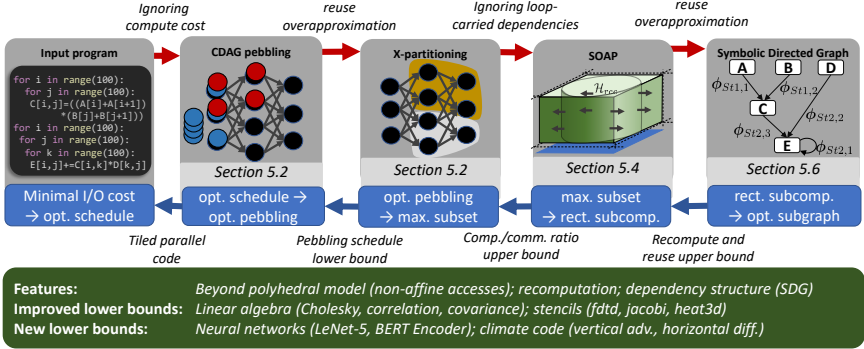


Figure 5.1: High level overview of the combinatorial SOAP analysis. An input program’s schedule is modeled as the red-blue pebble game. The X-Partitioning abstraction relaxes the pebbling problem to the graph partition problem. The SOAP abstraction utilizes the static loop structure to upper-bound the size of the optimal X-partition. The Symbolic Directed Graph (SDG) models inter-statement data dependencies. Our method derives I/O lower bounds together with accompanying tile sizes and loop fusions that can be used by a compiler to generate an I/O optimal parallel code.

inequalities [57–59] (a) are often too restrictive, requiring the programs to be expressed in the polyhedral model to count the points in the projection polytopes; (b) do not capture pebbling motifs such as recomputation [162]; or (c) are limited to single-statement programs [49, 57–59].

In our work, we take a different approach based on a combinatorial method. We directly map each elementary computation to a vertex in a parametric CDAG, which allows us not only to operate on unstructured iteration domains, but also to precisely count the sizes of dominator sets and model vertex recomputation. Furthermore, to handle complex data dependencies in programs that evaluate multiple arrays, we introduce the Symbolic Directed Graph (SDG) abstraction, which encapsulates the data flow between elementary computations. This allows us to cover a wider class of programs and handle more complex data flow.

To enable precisely mapping every data access to the parametric CDAG vertex, we extend the DAAP class of programs (Chapter 4) to *Simple Overlap Access Programs* (SOAP), and present a general method to derive *precise* I/O bounds of programs in this class. Specifically, SOAPS are defined as loop nests of statements, whose data access sets can be modeled as injective functions, and their per-statement data overlap can be expressed with constant offsets. For programs that do not directly adhere to SOAP,

with nontrivial overlaps and non-injective access functions, we show that under a set of assumptions, we can construct SOAP “projections” of those programs, which can be analyzed in the same way. Our method strictly contains the polyhedral model and associated analysis methods.

To show the breadth of our approach, we demonstrate SOAP analysis on a set of 38 applications, taking Python and C codes as input to create the SDG. This automated analysis procedure generates symbolic bounds, which match or improve upon previously-known results. Notably, we tighten the known I/O lower bounds for numerous programs, including stencils by up to a factor of 14, linear algebra kernels by a factor of two, and the core convolution operation in deep learning by a factor of 8.

Since our derivation of the bounds is constructive — i.e., it provides loop tilings and fusions after relaxing loop-carried dependencies — the results can be used by a compiler to generate I/O optimal parallel codes. This can both improve existing schedules and possibly reveal new parallelization dimensions.

The paper makes the following contributions:

- A combinatorial method for precisely counting the number of data accesses in parametric CDAGs.
- A class of programs — SOAP — on which I/O lower bounds can be automatically derived.
- Symbolic dataflow analysis that extends SOAP to multiple-statement programs, capturing input and output reuse between statements, as well as data recomputation.
- I/O analysis of 38 scientific computing kernels, improving existing bounds [162, 192] by up to a factor of 14, and new lower bounds for applications in deep learning, unstructured physics simulation, and numerical weather prediction.

5.2 BACKGROUND

In this chapter, we continue our work on modelling data movement. Specifically, we extend the DAAP model from Chapter 4 to capture overlapping accesses, as well as explicitly capture complex data reuse in programs containing multiple statements.

For the description of the program model (CDAG) and the memory model (the red-blue pebble game, the dominator and minimum sets) we

refer readers to the previous chapter. Here we only outline the definitions necessary for the SOAP analysis. The bird’s eye view of our method is presented in Figure 5.1.

5.2.1 General Approach of Modeling I/O Costs

Execution model: graph pebbling. An execution of a program represented by a CDAG $G = (V, E)$ is modeled as a sequence of four allowed pebbling moves: 1) placing a red pebble on a vertex which has a blue pebble (load), 2) placing a blue pebble on a vertex which has a red pebble (store), 3) placing a red pebble on a vertex whose parents have red pebbles (compute) 4) removing any pebble from a vertex (discard). At the program start, all input vertices have blue pebbles placed on them. Execution finishes when all output vertices have blue pebbles on them. A sequence of moves leading from the start to the end is called a graph *pebbling* P . The number of load and store moves in P is called the *I/O cost of P* . **The I/O cost Q of a program G is the minimum cost among all valid pebbling configurations.** A pebbling with cost Q is called optimal.

5.2.2 I/O Lower Bounds

Assume that the optimal pebbling P_{opt} is given. For any constant $X > S$ we can partition this sequence of moves into subsequences, such that in each subsequence except of the last one, exactly $X - S$ load/store moves are performed (the last subsequence contains at most $X - S$ load/store moves). Denote the number of these subsequences as h . Then observe that $(X - S)(h - 1) \leq Q \leq (X - S)h$ (Chapter 3).

Computational intensity. In previous chapters we proven that (a) Q is lower bounded by the number of subsequences h in the optimal pebbling P_{opt} ; (b) h is lower bounded by the size of the smallest X -partition $|\mathcal{P}_{min}(X)|$ for any value of $X > S$; (c) $|\mathcal{P}_{min}(X)|$ is bounded by the maximum size of a single subcomputation $|\mathcal{H}_{X,max}|$ in any valid X -partition: $|\mathcal{P}_{min}(X)| \geq |V|/|\mathcal{H}_{X,max}|$; and (d) if $|\mathcal{H}_{X,max}|$ can be expressed as a function of X , that is, $\chi(X) \equiv |\mathcal{H}_{X,max}|$, then Q is bounded by

$$Q \geq |V| \frac{X_0 - S}{\chi(X_0)}, \tag{5.1}$$

where $X_0 = \arg \min_X \frac{\chi(X)}{X-S}$ (Lemma 4.2). The expression $\rho = \frac{\chi(X)}{X-S}$ is called the *computational intensity*.

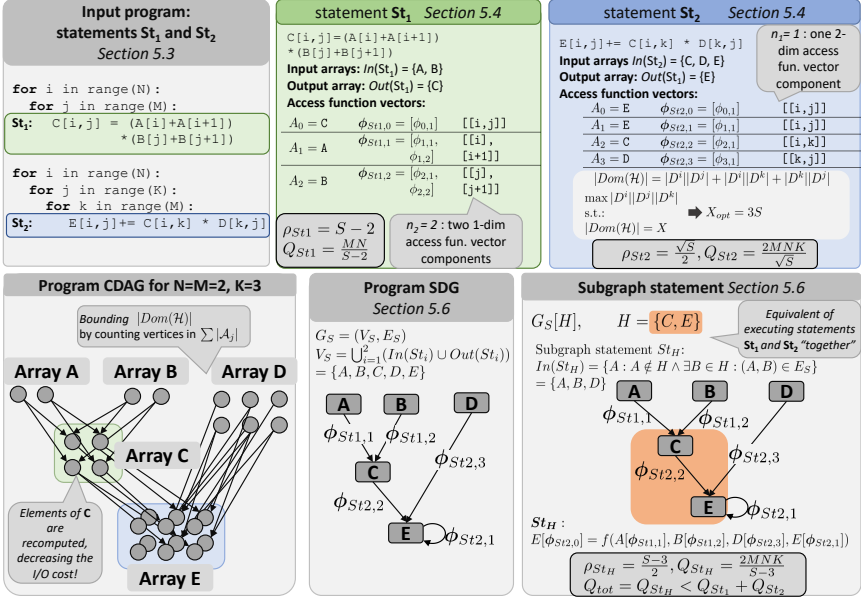


Figure 5.2: From the input code to the I/O lower bounds. First, for each statement, the access function vectors ϕ are extracted from the input program (green and blue fields). For each statement, the size of its dominator set is obtained using Lemma 5.3 (Section 5.4.2), and then, the I/O lower bound is obtained using inequality 5.9 (Section 5.4.5). For programs that contain multiple statements, the SDG is constructed (Section 5.5.3) and all valid subgraph statements are evaluated (Section 5.5.4). Lastly, the final I/O lower bound is obtained (Section 5.5.5).

5.3 SIMPLE OVERLAP ACCESS PROGRAMS

In previous chapters, we show how the I/O cost of a program can be bounded by the maximum size of a subcomputation \mathcal{H} in any valid X -partition of program CDAG. We now introduce **Simple Overlap Access Programs (SOAP)**: a class of programs for which we can derive tight analytic bounds of $|\mathcal{H}|$. We leverage the SOAP structure and design an end-to-end method for deriving I/O lower bounds of input programs (summarized in Figure 5.2).

What is SOAP? Before introducing the formal definition, we start with an illustrative example, which we use in the following sections.

Example 5.1. Consider the following 3-point stencil code (we use the Python syntax in code listings):

```

1  for t in range(1,T):
2    for i in range(t,N-t):
3      A[i,t+1]=(A[i-1,t] + A[i,t] + A[i+1,t])/3 + B[i]

```

This is what we will refer to as a *single-statement SOAP*. The program consists of one statement $St : A[i, t+1] = (A[i, t] + \dots)$ which is placed in two nested loops. All accessed data comes from static, disjoint, multi-dimensional arrays (A and B). Furthermore, different accesses to the same array (array A is referenced by $[i, t+1]$, $[i-1, t]$, $[i, t]$, $[i+1, t]$) are offset by a constant stride $[\theta, 1]$, $[-1, \theta]$, $[\theta, \theta]$, $[1, \theta]$. We denote such access pattern as a **simple overlap** and it is a defining property of SOAP.

Why SOAP? We use the restriction on the access pattern to precisely count the number of vertices in $Dom(\mathcal{H})$. If we allow arbitrary overlap of array accesses, we need to conservatively assume a maximum possible overlap of accessed vertices. This reduces the lower bound on $|Dom(\mathcal{H})|$, which, in turn, increases the upper bound on $|\mathcal{H}|$, providing less-tight I/O lower bound for a program.

This is not a fundamental limitation of our method. However, it allows a fully automatic derivation of tight I/O lower bounds for input programs. If the restriction is violated, additional assumptions on the access overlap are needed (Section 5.5).

SOAP definition. A program is a sequence of statements St_1, \dots, St_k . Each such statement St is a constant time computable function f enclosed in a loop nest of the following form:

$$\begin{aligned}
 &\text{for } \psi^1 \in \mathcal{D}^1 : \\
 &\quad \dots \\
 &\quad \text{for } \psi^\ell \in \mathcal{D}^\ell(\psi^1, \dots, \psi^{\ell-1}) : \\
 &\quad \quad St : A_0[\phi_0(\psi)] \leftarrow f(A_1[\phi_1(\psi)], A_2[\phi_2(\psi)], \dots, A_m[\phi_m(\psi)])
 \end{aligned}$$

where:

1. The *statement* St is nested in a loop nest of depth ℓ .
2. Each loop in the t th level, $t = 1, \dots, \ell$ is associated with its *iteration variable* ψ^t , which iterates over its domain $\mathcal{D}^t \subset \mathbb{N}$. Domain \mathcal{D}^t may depend on iteration variables from outer loops $\psi^1, \dots, \psi^{t-1}$ (denoted as $\mathcal{D}^t(\psi^1, \dots, \psi^{t-1})$).
3. All ℓ iteration variables form the *iteration vector* $\psi = [\psi^1, \dots, \psi^\ell]$ and we define the *iteration domain* \mathcal{D} as the set of all values the iteration vector iterates over during the entire execution of the program $\mathcal{D} \subset \mathbb{N}^\ell$.

4. The dimension of array A_j is denoted as $\dim(A_j)$.
5. Elements of A_j are referenced by an *access function vector* ϕ_j which maps $\dim(A_j)$ iteration variables $\psi_j = [\psi_j^1, \dots, \psi_j^{\dim(A_j)}]$ to a **set of n_j elements** from A_j , that is $\phi_j : \mathcal{D}_j^1 \times \dots \times \mathcal{D}_j^{\dim(A_j)} \rightarrow \left(\mathbb{N}^{\dim(A_j)}\right)^{n_j}$. We then write $\phi_j = [\phi_{j,1}, \dots, \phi_{j,n_j}]$, where $\phi_{j,k} : \mathcal{D}_j^1 \times \dots \times \mathcal{D}_j^{\dim(A_j)} \rightarrow \mathbb{N}^{\dim(A_j)}$, $k = 1, \dots, n_j$. Furthermore, all access function components $\phi_{j,k}(\psi_j)$ are injective.
6. All n_j access function vector's components are equal up to a constant translation vector, that is, $\forall k = 1, \dots, n : \phi_{j,k}(\psi) = \phi_{j,1}(\psi) + t_k$, where $t_k = [t_k^1, \dots, t_k^{\dim(A)}] \in \mathbb{N}^{\dim(A)}$. We call ϕ_j the **simple overlap access**.
7. Arrays A_1, \dots, A_m are disjoint. If the output array A_0 is also used as an input, that is, $A_0 \equiv A_j, j \geq 1$, then $\phi_0 \cup \phi_j$ is also the simple overlap access (c.f. Example 5.1).
8. Each execution of statement St is an evaluation of f for a given value of iteration vector ψ .

Iteration variables and iteration vectors. Formally, an iteration variable ψ^t is an iterator: an object which takes values from its iteration domain during the program execution. However, if it is clear from the context, we will refer to a particular *value* of the iteration variable simply as ψ^t (or a value of iteration vector as ψ).

Vertices as iteration vectors. Since by definition of CDAG, each computation corresponds to a different vertex, and by definition of SOAP, every statement execution is associated with a single iteration vector ψ , every non-input vertex in G is uniquely associated with an iteration vector ψ . Input vertices are referred to by their access function vectors $u = A_j[\phi_{j,k}(\psi)]$. **We further define CDAG edges as follows:** for every value of iteration vector ψ , we add an edge from all accessed elements to the vertex associated with ψ , that is: $E = \{(u, v) : u = A_j[\phi_{j,k}(\psi)], v = \psi, \psi \in \mathcal{D}\}$.

X-Partitioning on SOAP's CDAG. Recall that our objective is to bound the maximum size of any subcomputation $|\mathcal{H}|$. Given pebbling P and an associated X -partition $\mathcal{P}(X)$, every subcomputation $\mathcal{H} \in \mathcal{P}(X)$ is therefore associated with the set of iteration vectors ψ of the vertices computed in \mathcal{H} . In the following section we will derive it by counting how many non-input vertices (iteration vectors) can \mathcal{H} contain by bounding its dominator set

size $|Dom(\mathcal{H})|$ - again, by counting vertices corresponding to each access $A_j[\phi_{j,k}(\psi)]$.

SOAP definition (§ 5.3)	A_0	Output array of statement St (may overlap with input arrays).
	$A_j, j = 1, \dots, m$	Mutually disjoint input arrays of statement St .
	$\psi = [\psi^1, \dots, \psi^\ell]$	Iteration vector composed of ℓ iteration variables.
	$D \subseteq \mathcal{D}^1 \times \dots \times \mathcal{D}^\ell$	Iteration domain: a set of values that iteration vector ψ takes during the entire program execution.
	$\phi_j = [\phi_{j,1}, \dots, \phi_{j,n_j}]$	Access function vector that maps $dim(A_j)$ variables $[\psi_j^1, \dots, \psi_j^{dim(A_j)}]$ to n_j elements in array A_j .
$t_{j,k} = [t_{j,k}^1, \dots, t_{j,k}^{dim(A_j)}]$	Translation vector of k -th access function vector's component $\phi_{j,k}$, that is $\phi_{j,k} \equiv \phi_{j,1}, k = 1, \dots, n_j$	
Single-statement subcomputation (§ 5.4)	$\mathcal{P}(X) = \{\mathcal{H}_1, \dots, \mathcal{H}_s\}$	An X -partition of CDAG $G = (V, E)$ composed of s disjoint subcomputations.
	$D = \mathcal{D}^1 \times \dots \times \mathcal{D}^\ell$	Subcomputation domain: a Cartesian product of ranges of ℓ iteration variables during \mathcal{H} .
	$\mathcal{H} \subseteq D \subseteq V$	Subcomputation \mathcal{H} uniquely defined by a set of $ \mathcal{H} $ iteration vector's values $\psi \in D$ taken during \mathcal{H} . If $\mathcal{H} = D$, we call it a <i>rectangular subcomputation</i> \mathcal{H}_{rec} .
	$A = \phi[\mathcal{H}]$	Access set: a set of vertices from array A that are accessed by ϕ during \mathcal{H} .
	$\hat{t}^i = \{t_1^i, \dots, t_n^i\} \setminus \{0\}$	Access offset set: set of all non-zero i th coordinates among n translation vectors $t_k, k = 1, \dots, n$.
	$Dom(\mathcal{H})$	Dominator set of subcomputation \mathcal{H} .
	ρ	The computational intensity of the X -partition.
$Q \geq \mathcal{D} \frac{\sum_{j=1}^m A_j(X_0) - S}{\prod_{i=1}^\ell D^i(X_0) }$	A number of I/O operations of a schedule.	
SDG (§ 5.5.3)	$G_S = (V_S, E_S)$	Symbolic Directed Graph, where every array accessed in a program is a vertex, and edges represent data dependencies between them.
	$I \subset V_S$	Set of read-only arrays of the program.
	$G_S[H], H \subset V_S \setminus I$	SDG subgraph that represents a subcomputation in which at least one vertex from every array in H is computed.
	St_H	Subgraph SOAP statement.

Table 5.1: Notation used in this chapter.

5.4 I/O LOWER BOUNDS FOR SINGLE-STATEMENT SOAP

We now derive the I/O bounds for programs that contain only one SOAP statement. We start with introducing necessary definitions that allow us to bound the size of a *rectangular subcomputation*. The summary of the notation is presented in Table 5.1.

5.4.1 Definitions

Definition 5.1. Subcomputation domain. Denote the set of all values which iteration variable ψ^t takes during subcomputation \mathcal{H} as $D^t \subset \mathcal{D}^t, t = 1, \dots, \ell$. Then, the **subcomputation domain** $\mathbf{D}(\mathcal{H}) \subseteq \mathcal{D}$ is a Cartesian product of ranges of all ℓ iteration variables which they take during \mathcal{H} , that is $\mathbf{D}(\mathcal{H}) = D^1 \times \dots \times D^\ell$. We therefore have $\mathcal{H} \subseteq \mathbf{D}(\mathcal{H}) \subset \mathbb{N}^\ell$. If it is clear from the context, we will sometimes denote $\mathbf{D}(\mathcal{H})$ simply as \mathbf{D} .

Example 5.2. Recall the program from Example 5.1. Consider subcomputation \mathcal{H} in which $\mathfrak{t} \in \{1, 2\}$ and $\mathfrak{i} \in \{1, 2\}$. Then, subcomputation domain $\mathbf{D} = \{1, 2\} \times \{1, 2\} = \{[1, 1], [1, 2], [2, 1], [2, 2]\}$, but computation itself can contain at most 3 elements $\mathcal{H} \subseteq \{[1, 1], [1, 2], [2, 2]\}$, since $\psi = [2, 1] \notin \mathcal{D}$ does not belong to the iteration domain.

Definition 5.2. Access set and access subdomain. Consider input array A and its access function vector ϕ . Given \mathcal{H} , the **access set** \mathcal{A} of A is the set of vertices belonging to A that are accessed during \mathcal{H} , that is $\mathcal{A} = \phi[\mathcal{H}] = \{A[\phi(\psi)] : \psi \in \mathcal{H}\}$. If function $\phi = [\phi_1, \dots, \phi_n]$ accesses n vertices from A , we analogously define access sets for each access function component $\phi_k[\mathcal{H}], k = 1, \dots, n$. We then have $\mathcal{A} = \bigcup_{k=1}^n \phi_k[\mathcal{H}]$. The **access subdomain** $\mathbf{D}(\mathcal{A})$ is minimum bounding box of the access set \mathcal{A} .

Example 5.3. For program in Example 5.1, consider subcomputation \mathcal{H} evaluated on only one iteration vector $\mathcal{H} = [i = 2, j = 2]$. We have two accessed arrays A and B . Furthermore, we have $\phi_A = [[i, t + 1], [i - 1, t], [i, t], [i - 1, t]]$. Therefore, $\dim(A) = 2$, and $\phi_B : \mathbb{N}^2 \rightarrow (\mathbb{N}^2)^4$. We further have $\phi_B = [[i]]$, $\dim(B) = 1$, and $\phi_A : \mathbb{N} \rightarrow \mathbb{N}$. To evaluate St for $\psi = [2, 2]$, we need to access four elements of A (three loads and one store), so its access set is $\mathcal{A} = \phi_A[\mathcal{H}] = \{[2, 3], [1, 2], [2, 2], [2, 3]\}$. Furthermore, we have the access subdomain $\mathbf{D}(\mathcal{A}) = \{2, 3\} \times \{1, 2, 3\}$.

Definition 5.3. Access offset set. Given a simple overlap access $\phi = [\phi_1, \dots, \phi_n]$ consider its n translation vectors $t_k = [t_k^1, \dots, t_k^{\dim(A_j)}] \in$

$\mathbb{N}^{\dim(A)}$, $k = 1, \dots, n$. For each dimension $i = 1, \dots, \dim(A_j)$ we denote $\hat{t}^i = \{t_1^i, \dots, t_n^i\} \setminus \{0\}$ as the set of all unique non-zero i th coordinates among all n translation vectors.

Definition 5.4. Rectangular subcomputation For a given subcomputation domain \mathbf{D} , a subcomputation \mathcal{H} is called **rectangular** if $\mathcal{H} = \mathbf{D}$ and is denoted $\mathcal{H}_{rec}(\mathbf{D})$. The size of rectangular computation is $|\mathcal{H}_{rec}(\mathbf{D})| = \prod_{t=1}^{\ell} |D^t|$. If it is clear from the context, we will denote $\mathcal{H}_{rec}(\mathbf{D})$ simply as \mathcal{H}_{rec} .

Observation 1. Consider a simple overlap access $\boldsymbol{\phi} = [\boldsymbol{\phi}_1, \dots, \boldsymbol{\phi}_n]$ of array A and a rectangular subcomputation $\mathcal{H}_{rec}(\mathbf{D})$. Then since all $\boldsymbol{\phi}_k$ are equal up to translation, the ranges of iteration variables they access are also equal up to the same translation: $\forall i = 1, \dots, \dim(A) : \forall j = 1, \dots, n : \boldsymbol{\phi}_j[D^i] = \boldsymbol{\phi}_1[D^i] + t_j$, which also implies that $\forall i = 1, \dots, \dim(A) : \forall j = 1, \dots, n : |\boldsymbol{\phi}_j[D^i]| = |\boldsymbol{\phi}_1[D^i]|$.

To bound the sizes of rectangular subcomputations, we need Lemmas 5.1 and 5.2 from Chapter 4. Since SOAP extends the DAAP model with overlapping accesses and adds new definitions, here we reintroduce them using the notation from Section 5.4.1:

Lemma 5.1. For statement St , given \mathbf{D} , the size of subcomputation \mathcal{H} (number of vertices of S computed during \mathcal{H}) is bounded by the sizes of the iteration variables' sets D^t , $t = 1, \dots, \ell$:

$$|\mathcal{H}| \leq \prod_{t=1}^{\ell} |D^t|. \quad (5.2)$$

Lemma 5.2. For the given access function $\boldsymbol{\phi} = [\boldsymbol{\phi}_1, \dots, \boldsymbol{\phi}_n]$ accessing array A , $A[\boldsymbol{\phi}(\boldsymbol{\psi})]$, the access set size of each of components $|\boldsymbol{\phi}_k[\mathcal{H}]|$ during subcomputation \mathcal{H} is bounded by the sizes of $\dim(\boldsymbol{\phi}_A)$ iteration variables' sets D^i , $k = 1, \dots, \dim(\boldsymbol{\phi}_j)$:

$$|\boldsymbol{\phi}_k[\mathcal{H}]| \leq \prod_{i=1}^{\dim(\boldsymbol{\phi}_A)} |D^i| \quad (5.3)$$

where $D^i \ni \psi^i$ is the iteration domain of variable ψ^i during \mathcal{H} .

5.4.2 Bounding SOAP Access Size

Recall that our goal is to find the maximum size of the subcomputation given its dominator size. We first do the converse: given the rectangular subcomputation \mathcal{H}_{rec} , we bound the minimum number of input vertices

required to compute \mathcal{H}_{rec} . In Section 5.4.4 we prove that indeed \mathcal{H}_{rec} is the subcomputation that upper-bounds the maximum computational intensity ρ . Since arrays A_1, \dots, A_m are disjoint, the total number of input vertices is the sum of their access set sizes: $|Dom_{min}(\mathcal{H}_{rec})| \geq \sum_{j=1}^m |\mathcal{A}_j|$. We now proceed to bound individual access set sizes $|\mathcal{A}_j|$.

Consider array A with $dim(A) = d$ and its access function $\phi(\psi) = [\phi_1(\psi), \dots, \phi_n(\psi)]$ that access n elements from A (to simplify the notation, we drop the subscript j , since we consider only one array). Observe that during \mathcal{H}_{rec} , all combinations of iteration variables $\psi^1 \in D^1, \dots, \psi^\ell \in D^\ell$ are accessed, so $|\mathcal{H}_{rec}| = \prod_{t=1}^\ell |D^t|$ (Lemma 5.1). This also implies that each of $k = 1, \dots, n$ accesses to A required $|\phi_k[\mathcal{H}_{rec}]| = \prod_{t=1}^d |D^t|$ vertices from A (Lemma 5.2 and Observation 1). Therefore, the total number of accesses to array A during \mathcal{H}_{rec} is $|\mathcal{A}| \geq \prod_{t=1}^d |D^t|$. However, the sets of vertices accessed by different ϕ_k may overlap, that is, there may exist two accesses ϕ_l and ϕ_m , for which $\phi_l[\mathcal{H}_{rec}] \cap \phi_m[\mathcal{H}_{rec}] \neq \emptyset$. Therefore, we also obtain the upper bound $|\mathcal{A}| \leq \sum_{j=1}^n \prod_{t=1}^d |D^t|$. We now want to narrow the gap between the upper and the lower bounds.

Lemma 5.3. *If a given input array A with $dim(A) = d$ is accessed by a simple overlap access $\phi(\psi) = [\phi_1(\psi), \dots, \phi_n(\psi)]$, its access set size $|\mathcal{A}|$ during rectangular computation $\mathcal{H}_{rec}(D)$ is bounded by*

$$|\mathcal{A}| = |\phi[\mathcal{H}_{rec}(D)]| \geq 2 \prod_{i=1}^d |D^i| - \prod_{i=1}^d (|D^i| - |\hat{t}^i|), \tag{5.4}$$

where $|\hat{t}^i|$ is the size of the access offsets set in the i th dimension.

Proof. W.l.o.g., consider the first access function component ϕ_1 and its $\prod_{t=1}^d |D^t|$ accessed vertices $\phi_1[\mathcal{H}_{rec}]$. We will lower bound the number of accesses to A from remaining $\phi_k, k = 2, \dots, n$, which do not overlap with $\phi_1[\mathcal{H}_{rec}]$, that is $|\bigcup_{k=2}^n \phi_k[\mathcal{H}_{rec}] \setminus \phi_1[\mathcal{H}_{rec}]|$. Since by construction of \mathcal{H}_{rec} , all $\phi_k[\mathcal{H}_{rec}]$ are Cartesian products of iteration variables' ranges $\phi_k[D^1] \times \dots \times \phi_k[D^d]$, there is a bijection between $\phi_k[\mathcal{H}_{rec}]$ and an d -dimensional hyperrectangle $H_k \in \mathbb{N}^d$. To secure correctness of our lower bound on $|\mathcal{A}|$, we need to find the volume of the smallest union of these hyperrectangles.

Note that $|\hat{t}^i|$ is a lower bound on the maximum offset between any two $H_j \neq H_k$ in dimension i : the union of all hyperrectangles $\bigcup_{k=1}^n H_k$ "stretches" at least $|D^i| + |\hat{t}^i|$ elements in the i th dimension for all $i = 1, \dots, d$ (see Figure 5.3). To see this, observe that since $D^i \subset N$, for each element in the access offset set $t_j^i \in \hat{t}^i$ there is at least one element in $D^i + t_j^i$ that is not in D^i , which implies that $|(D^i + t_j^i) \setminus D^i| \geq 1$. Since D^i is finite, there is a

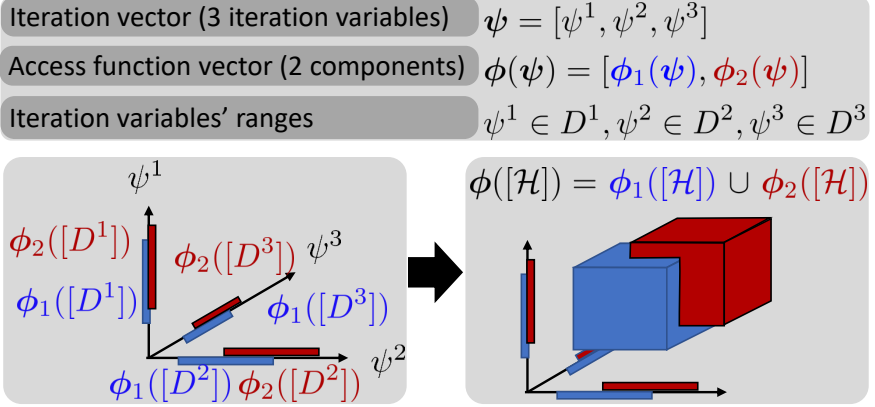


Figure 5.3: Intuition behind Lemma 5.3. Access sets $\phi[\mathcal{H}_{rec}(D)]$ as 3-dimensional hyperrectangles. The union $|\bigcup_{k=1}^n \phi_k[\mathcal{H}_{rec}]|$ (and therefore, the total number of accesses $|\mathcal{A}|$) is minimized when the hyperrectangles are placed in two antipodal locations of the subcomputation domain D .

single well-defined maximum and a minimum element, which implies that $(\max\{D^i\} + t_j^i \notin D^i) \vee (\min\{D^i\} + t_j^i \in D^i)$. Also, because by definition of \hat{t}^i we have $\forall t_j^i, t_k^i \in \hat{t}^i : t_j^i \neq t_k^i$, then we also have that each t_j^i accesses at least one “non-overlapping” element independent of any other t_k^i , that is $\forall t_j^i, t_k^i \in \hat{t}^i : \max\{D^i\} + t_j^i \neq \max\{D^i\} + t_k^i$.

The arrangement of hyperrectangles $H_k, k = 1, \dots, n$ in a \mathbb{N}^d lattice s.t., their bounding box is $D = (|D^1| + |\hat{t}^1|) \times \dots \times (|D^d| + |\hat{t}^d|)$, which minimizes the size of their union $|\bigcup_k H_k|$ satisfies two properties:

1. there exist two “extreme” H_p, H_q , such that $H_q = H_p + v$, $u = \mathbb{Z}^d, \forall_{i=1, \dots, d} : |v^i| = |\hat{t}^i|$,
2. all the remaining $H_k, k \neq p, q$ perfectly overlap with the “extreme” hyperrectangles $H_k \subseteq H_p \cup H_q$.

To see this, observe that for every non-zero $|\hat{t}^i|$ we need two hyperrectangles $H_p^i \neq H_q^i$ s.t., $H_q^i = H_p^i + [\cdot, \dots, |\hat{t}^i|, \dots, \cdot]$, that is, H_q^i is offset from H_p^i by $|\hat{t}^i|$ in i th dimension. We therefore have $\bigcup_{i, |\hat{t}^i| > 0} (H_p^i \cup H_q^i) \subseteq \bigcup_k H_k$. Since H_p^i and H_q^i are pairwise non-equal, but there are no restrictions on $H_p^i, H_p^j, i \neq j$, we have that the union $\bigcup_{i, |\hat{t}^i| > 0} (H_p^i \cup H_q^i)$ is minimized if $\forall_{i \neq j} H_p^i = H_p^j$.

Finally, observe the volume of $|\bigcup_{k=1}^n H_k|$ s.t. to the claimed arrangement is:

$$\left| \bigcup_{k=1}^n H_k \right| = |H_p \cup H_q| = 2 \prod_{i=1}^d |D^i| - \prod_{i=1}^d (|D^i| - |\hat{t}^i|) \quad (5.5)$$

It shows that for any set of n hyperrectangles s.t. the given constraint, the volume of their union is no smaller than the one in Equation 5.5. Since the offset constraint is also a lower bound on the number of non-overlapping accesses in each dimension, it also forms the bound on $|\bigcup_{k=1}^n \phi_k[\mathcal{H}_{rec}]| = |\phi[\mathcal{H}_{rec}]| = |\mathcal{A}|$. \square

5.4.3 Input-Output Simple Overlap

If one of the input arrays $A_i, i \geq 1$, is also the output array A_0 , then their access function vectors ϕ_0 and ϕ_i form together a simple overlap access (Section 5.3). In such cases, some vertices accessed by ϕ_i during \mathcal{H}_{rec} may be computed and do not need to be loaded. We formalize it in the following corollary, which follows directly from Lemma 5.3:

Corollary 5.1. *Consider statement St that computes array A , $\dim(A) = d$ and simultaneously accesses it as an input $A[\phi_0(\psi)] = f(A[\phi_1(\psi)])$. If $\phi_0 \cup \phi_1$ is a simple overlap access, the access set size $|\mathcal{A}|$ during rectangular computation \mathcal{H}_{rec} is bounded by*

$$|\mathcal{A}| \geq \prod_{i=1}^d |D^i| - \prod_{i=1}^d (|D^i| - |\hat{t}^i|), \quad (5.6)$$

where \hat{t} is an access offset set of $\phi_0 \cup \phi_1$.

Proof. This result follows directly from Lemma 5.3. Since there are at least $2 \prod_{i=1}^d |D^i| - \prod_{i=1}^d (|D^i| - |\hat{t}^i|)$ vertices accessed from A_i , and at most $\prod_{i=1}^d |D^i|$ of them can be computed during \mathcal{H}_{rec} (Lemma 5.2) and therefore, do not have to be loaded, then at least $2 \prod_{i=1}^d |D^i| - \prod_{i=1}^d (|D^i| - |\hat{t}^i|) - \prod_{i=1}^d |D^i|$ elements have to be accessed from the outside of \mathcal{H}_{rec} . \square

5.4.4 Bounding Maximal Subcomputation

In Section 5.4.2 we lower-bounded the dominator set size of the rectangular subcomputation $|Dom_{min}(\mathcal{H}_{rec})| = \sum_{j=1}^m |\mathcal{A}_j|$ by bounding the sizes of simple overlap access sets sizes $|\mathcal{A}_j|$ (Lemma 5.3). Recall that to bound the I/O

lower bound we need the size $\chi(X)$ of the *maximal* subcomputation \mathcal{H}_{max} for given value of X (Inequality 5.1). We now prove that \mathcal{H}_{rec} upper-bounds the size of \mathcal{H}_{max} .

Given \mathcal{H} , denote the ratio of the size of the subcomputation to the dominator set size $\delta(\mathcal{H}) = \frac{|\mathcal{H}|}{\sum_{j=1}^m |\phi_j[\mathcal{H}]|}$. By definition, \mathcal{H}_{max} maximizes δ among all valid $\mathcal{H} \in \mathcal{P}$. We need to show that for a fixed subcomputation domain D_0 , among all subcomputations for which $D(\mathcal{H}) = D_0$, the rectangular subcomputation $\mathcal{H}_{rec}(D_0)$ upper-bounds δ . Note that an X -partition derived from the optimal pebbling schedule P_{opt} may not include \mathcal{H}_{rec} . However, $\forall X : \chi_{rec}(X) \geq \chi(X)$, that is, given X , the size of \mathcal{H}_{rec} s.t., $\sum_{j=1}^m |\phi_j[\mathcal{H}_{rec}]| = X$ will always be no smaller than the size of \mathcal{H}_{max} . To show this, we first need to introduce some auxiliary definitions.

Iteration variables, their indices, and their values. To simplify the notation, throughout the paper we used the iteration variables ψ^i and the *values* they take for some iteration interchangeably. However, now we need to make this distinction explicit. The iteration vector consists of ℓ iteration variables $\boldsymbol{\psi} = [\psi^1, \dots, \psi^\ell]$. Each access function ϕ_j is defined on $dim(A_j) \leq \ell$ of them. Recall that Ψ_j is the set of iteration variables accessed by ϕ_j (Section 5.3, property (5)). To keep track of the indices of particular iteration variables, denote $\mathbf{\Psi} = [\ell] = \{1, \dots, \ell\} \subset \mathbb{N}$, $\mathbf{\Psi}_j \subseteq \mathbf{\Psi}$, and $\mathbf{\Psi}'_j = \mathbf{\Psi} \setminus \mathbf{\Psi}_j$ as the sets of integers. If $i \in \mathbf{\Psi}_j$, then the i th iteration variable ψ^i is accessed by the access function ϕ_j . We further define $\boldsymbol{\psi}^* \in \mathbb{N}^\ell$ as a specific *value* of the iteration vector $\boldsymbol{\psi}$ that uniquely defines a single non-input vertex. We analogously define $\boldsymbol{\psi}_j^*$, $\psi^{i,*}$, and $\psi_j^{i,*}$ (the last one being a value of i th iteration variable of the j th access). We also define $\theta(\boldsymbol{\psi}_j^*, \mathcal{H})$ as the number of vertices in \mathcal{H} that have all their $\mathbf{\Psi}_j$ coordinates equal to $\boldsymbol{\psi}_j^*$, that is $\theta(\boldsymbol{\psi}_j^*, \mathcal{H}) = |\{\boldsymbol{\psi}^* : \boldsymbol{\psi}^* \in \mathcal{H} \wedge (\forall i \in \mathbf{\Psi}_j : \psi^{i,*} = \psi_j^{i,*})\}|$.

We now formalize our claim in the following lemma:

Lemma 5.4. *Given the subcomputation domain D_0 , $\mathcal{H}_{rec}(D_0)$ maximizes $\delta(\mathcal{H})$ for all \mathcal{H} s.t. $D(\mathcal{H}) = D_0$.*

$$\forall \mathcal{H} : \delta(\mathcal{H}) \leq \delta(\mathcal{H}_{rec}) \quad (5.7)$$

Proof. Instead of maximizing $\delta(\mathcal{H})$, we will minimize $\delta^{-1}(\mathcal{H}) = (\sum_{j=1}^m |\phi_j[\mathcal{H}]|) / |\mathcal{H}| = \sum_{j=1}^m |\phi_j[\mathcal{H}]| / |\mathcal{H}|$ over all possible \mathcal{H} . Observe that $\delta^{-1}(\mathcal{H})$ is linear w.r.t. the ratios of individual access function sets sizes $|\phi_j[\mathcal{H}]|$ and the size of subcomputation $|\mathcal{H}|$. Therefore, we can examine each access $\phi_j[\mathcal{H}]$ separately and show that every $\delta_j^{-1} = |\phi_j[\mathcal{H}]| / |\mathcal{H}|$

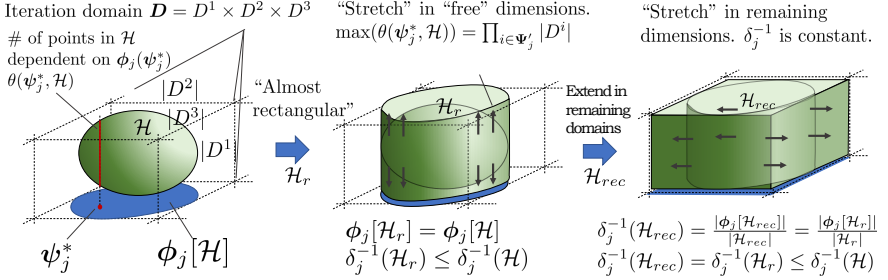


Figure 5.4: Intuition behind Lemma 5.4: extending the subcomputation in the free dimensions w.r.t ϕ_j does not increase $|\phi_j[\mathcal{H}]|$. Once the subcomputation is almost rectangular, extending H in the remaining dimensions keeps the ratio δ_j^{-1} constant.

is minimized for $\mathcal{H} = \mathcal{H}_{rec}$. Then, if \mathcal{H}_{rec} minimizes each of δ_j^{-1} , then $\delta^{-1} = \sum_{j=1}^m \delta_j^{-1}$ is minimized, so indeed \mathcal{H}_{rec} maximizes the ratio of the subcomputation size to the dominator set size.

Observe now, that for any \mathcal{H} we have that $\forall_j : \delta_j^{-1}$ is monotonically decreasing w.r.t. $\theta(\psi_j^*, \mathcal{H})$ for all $\psi_j^* \in \phi_j[\mathcal{H}]$. That is - pick any input vertex ψ_j^* from the set of vertices accessed by $\phi_j[\mathcal{H}]$. Adding compute vertices ψ^* to \mathcal{H} that access ψ_j^* do not increase the access set size $\phi_j[\mathcal{H}]$, since ψ_j^* is already accessed. However, it increases the size of \mathcal{H} . Clearly, δ_j^{-1} reaches its minimum if $\forall \psi_j^* \in \phi_j[\mathcal{H}] : \theta(\psi_j^*, \mathcal{H}) = \prod_{i \in \Psi'_j} |D^i|$, that is, \mathcal{H} computes all vertices spanned by the access set $\phi_j[\mathcal{H}]$ and all elements in the Cartesian product of “free” (independent of the access function ϕ_j) iteration domains $D^i, i \in \Psi'_j$.

We showed that for all j , given its initial access set $\phi_j[\mathcal{H}]$, the ratio δ_j^{-1} is minimized for the “almost-rectangular” subcomputation, that is, \mathcal{H} which computes all vertices $\psi^* \in \phi_j[\mathcal{H}] \times \prod_{i \in \Psi'_j} D^i$. We now need to show that also extending \mathcal{H} over the “dependent” ranges Ψ_j won’t increase the ratio δ^{-1} . When the access set size $\phi_j[\mathcal{H}]$ increases by a factor x , \mathcal{H} increases proportionally by x too, keeping the ratio constant (See Figure 5.4 for an example for $\ell = 3$).

Since our goal is to minimize each δ_j^{-1} separately, independently of other $\delta_i^{-1}, i \neq j$, assume that we have already extended \mathcal{H} to the “almost-rectangular” subcomputation, that is, all combinations of $\prod_{i \in \Psi'_j} D^i$ were

accessed in \mathcal{H} . Observe now that $\theta(\boldsymbol{\psi}_j^*, \mathcal{H}) = \prod_{i \in \Psi_j'} |D^i|$ for any vertex $\boldsymbol{\psi}_j^*$. Therefore, since $|\mathcal{H}| = \sum_{\boldsymbol{\psi}_j^* \in \Phi_j[\mathcal{H}]} \prod_{i \in \Psi_j'} |D^i|$, we see that δ_j^{-1} is constant w.r.t., the size of the access set: $\delta_j^{-1} = \frac{|\Phi_j[\mathcal{H}]|}{|\mathcal{H}|} = \frac{1}{\prod_{i \in \Psi_j'} |D^i|}$. Therefore, we can safely maximize $\Phi_j[\mathcal{H}]$ to the entire access set of the rectangular subcomputation \mathcal{H}_{rec} without increasing δ_j^{-1} . We conclude that for every access function Φ_j and every iteration variable index i , evaluating all vertices $\boldsymbol{\psi}^*$ s.t. $\boldsymbol{\psi}^i$ iterates over the entire domain D^i minimizes δ_j^{-1} . \square

5.4.5 I/O Lower Bounds and Optimal Tiling

We now proceed to the final step of finding the I/O lower bound. Recall from Section 5.2.2, that the last missing piece is $\chi(X)$; that is, we seek to express $|\mathcal{H}_{max}(\mathbf{D})| = \prod_{t=1}^{\ell} |D^t|$ as a function of X . Observe that by Lemma 5.4, $|\text{Dom}_{min}(\mathcal{H}_{max}(\mathbf{D}))| \geq \sum_{j=1}^m (2 \prod_{i=1}^{dim(A_j)} |D_j^i| - \prod_{i=1}^{dim(A_j)} (|D_j^i| - |\hat{t}_j^i|))$. On the other hand, by definition of X -Partitioning, $|\text{Dom}_{min}(\mathcal{H}_{max}(\mathbf{D}))| \leq X$. Combining these inequalities, we solve for all $|D^t|$ as functions of X by formulating it as the optimization problem (Section 4.3.2):

$$\begin{aligned} & \max \prod_{t=1}^{\ell} |D^t| \quad \text{s.t.} \\ & \sum_{j=1}^m |\mathcal{A}_j| \leq X \\ & \forall 1 \geq t \geq \ell : |D^t| \geq 1 \end{aligned} \quad (5.8)$$

Solving the above optimization problem yields $\chi(X) = |\mathcal{H}_{max}(\mathbf{D})|$. Since Lemma 5.4 gives a valid upper bound on computational intensity for any value of X , we seek to find the tightest (lowest) upper bound. One can obtain $X_0 = \arg \min_X \frac{\chi(X)}{X-S}$, since $\chi(X)$ is differentiable. Finally, combining Lemma 5.3, inequality 5.1, and the optimization problem 5.8, we obtain the I/O lower bound for the single-statement SOAP program:

$$Q \geq |\mathbf{D}| \frac{\sum_{j=1}^m |\mathcal{A}_j(X_0)| - S}{\prod_{t=1}^{\ell} |D^t(X_0)|}, \quad (5.9)$$

where $|A_j(X_0)|$ are the access set sizes obtained from Lemma 5.3 for the optimal value of $|D^t|$ derived from the optimization problem 5.8.

Substituting X_0 back to $|D^t|(X)$ has a direct interpretation: they constitute optimal loop tilings for the maximal subcomputation. Note that such tiling might be invalid due to problem relaxations: e.g., we ignore loop-carried dependencies and we solve optimization problem 5.8 over real numbers, relaxing the integer constraint on $|D^t|$ set sizes. *However, this result can serve as a powerful guideline in code generation. Furthermore, if derived tiling sizes generate a valid code, it is provably I/O optimal.*

5.5 PROJECTING PROGRAMS ONTO SOAP

By the definition of SOAP, one input array may be accessed by different access function vector components, only if they form the simple overlap access — that is, the accesses are offset by a constant stride. However, our analysis may go beyond this constraint if additional assumptions are met.

5.5.1 Non-Overlapping Access Sets

Given input array A and its access function components $\phi(\psi) = [\phi_1(\psi_1), \dots, \phi_n(\psi_n)]$, if all access sets are disjoint, that is: $\forall_{i \neq j} \phi_i[\mathcal{D}] \cap \phi_j[\mathcal{D}] = \emptyset$, then we represent it as n disjoint input arrays A_i accessed by single corresponding access function component $\phi_i(\psi_i)$.

Example 5.4. Consider the following code fragment from LU decomposition:

```

1  for k in range(N):
2    for i in range(k+1,N):
3      for j in range(k+1,N):
4  St:      A[i,j] = A[i,j] - A[i,k] * A[k,j]
```

The analysis of iteration variables' domains $\mathcal{D}^i, \mathcal{D}^j, \mathcal{D}^k$ shows that for fixed value of k_0 , there are no two iteration vectors $\psi_1 = [k_0, i_1, j_1]$ and $\psi_2 = [k_0, i_2, j_2]$ such that $[i_1, k_0] = [k_0, j_2] \vee [i_1, j_1] = [k_0, j_2] \vee [i_1, j_1] = [i_1, k_0]$, therefore, their access sets are disjoint. Furthermore, for k_0 , all elements from A in range $[(k_0, N), (k_0, N)]$ are updated. Therefore, all accesses of form $[i_1, k_1] = [k_2, j_2]$ access different vertices. We model this as a SOAP statement with three disjoint arrays:

$$St_2 : A_1[i, j] = f(A_1[i, j], A_2[i, k], A_3[k, j])$$

5.5.2 Equivalent Input-Output Accesses

If array A is updated by statement St — i.e., it is both input and output — then we require that the output access function ϕ_0 is different than the input access function ϕ_i . If the input program does not meet this requirement, we can add additional “version dimension” to access functions that is offset by a constant between input and output accesses.

Example 5.5. Consider again Example 5.4. Observe that array A_1 is updated (it is both the input and the output of St_2). Furthermore, both access functions are equal: $\phi_0 = \phi_1 = [i, j]$. We can associate a unique version (and therefore, a vertex) of each element of A with a corresponding iteration of the k loop. We add the version dimension associated with k and offset it by constant 1 between input and output:

$$St_3 : A_1[i, j, k + 1] = f(A_1[i, j, k], A_2[i, k], A_3[k, j])$$

5.5.3 Non-Injective Access Functions

Given input array A and its access function vector ϕ , we require that $\forall \psi_i \neq \psi_j : A[\phi(\psi_i)] \neq A[\phi(\psi_j)]$. If this is not the case, then we seek to bound the size of such overlap, that is, given subcomputation domain $D(\mathcal{H})$, how many different iteration vectors ψ_j map to the same array element $A[\phi(\psi_i)]$. We can solve this by analyzing the iteration domain \mathcal{D} and the access function vector ϕ . If one array dimension is accessed by a function of multiple iteration variables $g(\phi^1, \dots, \phi^k)$ and g is linear w.r.t. all ϕ^i , the number of different values g takes in $D(\mathcal{H})$ is bounded by $\max_{i=1, \dots, k} \{|D^i|\} \leq |g[\mathcal{H}]| \leq \prod_{i=1}^k |D^i|$, for $D^i \neq \{0\}, i = 1, \dots, k$.

Example 5.6. A single layer of the direct convolution used in neural networks may be written as seven nested loops with iteration variables b, c, k, w, h, r, s and statement (c.f. [59]):

$$St : Out[k, h, w, b] += Image[r + \sigma_w w, s + \sigma_h h, c, b] \times Filter[k, r, s]$$

Depending on the value of σ_w and σ_h , the access function of $Image$, $\phi = [r + \sigma_w w, s + \sigma_h h, c, b]$ may not be injective. Yet, observe that:

$$1. \sigma_w \geq |D^r| \wedge \sigma_h \geq |D^s| \implies \phi \text{ is injective} \implies |\phi[\mathcal{H}_{max}]| \geq |D^r| \cdot |D^w| \cdot |D^s| \cdot |D^h| \cdot |D^c| \cdot |D^b|$$

$$2. \sigma_w = 1 \wedge \sigma_h = 1 \implies |\phi[\mathcal{H}_{max}]| \geq \max(|D^r|, |D^w|) \cdot \max(|D^s|, |D^h|) \cdot |D^c| \cdot |D^b|,$$

Our analysis provides a conditional computational intensity: $\rho_{min} = \sqrt{S}/2$ in case (1) and $\rho_{max} = S/2$ in case (2). Observe that case (2) yields the maximum non-injective overlap (maximum number of different iteration vectors map to the same element in Image). For any other values of σ_w and σ_h , we have $\rho_{min} \leq \rho \leq \rho_{max}$.

I/O lower bounds are not composable: the I/O cost of a program containing multiple statements may be lower than the sum of the I/O costs of each statement if evaluated in isolation. Data may be reused and merging of statements may lower the I/O cost.

Note that the number of vertices in the program's CDAG G depend on domain sizes D^i of each iteration variable. However, our derived upper bound of the computational intensity ρ is *independent* of the CDAG size, as it depends only on the access functions ϕ_j . This is also true for programs that contain multiple statements - to bound ρ for multi-statement SOAP, we only need to model dependencies between the arrays and how they are accessed - e.g., one statement may take as an input an array that is an output of a different statement.

We represent the data flow between the program statements with a *symbolic* directed graph $G_S = (V_S, E_S)$. For a given statement St_i , denote $In(St_i) = \{A_{i,1}, \dots, A_{i,m}\}$ a set of input arrays of statement St_i . Analogously, denote $Out(St_i)$ the set containing the output array of St_i . Analogously to program CDAG G that captured dependencies between particular array elements, G_S models dependencies between whole arrays (Figure 5.2).

Definition 5.5. Symbolic Digraph: SDG Given k -statement SOAP St_1, \dots, St_k , its symbolic digraph (SDG) $G_S = (V_S, E_S)$ is a directed graph where $V_S = \bigcup_{i=1}^k (In(St_i) \cup Out(St_i))$ and $(A_u, A_v) \in E_S \iff \exists St_i : A_u \in In(St_i) \wedge A_v \in Out(St_i)$.

G_S is a directed graph, where vertices represent arrays accessed by a program, and edges represent data dependencies between them. Two arrays A_u and A_v are connected if there is a statement that accesses A_u and computes A_v . Each edge is annotated with the corresponding access function vector of the statement that generates it.

Example 5.7. Consider the example in Figure 5.2. We have two statements St_1 and St_2 , with $In(St_1) = \{A, B\}$, $Out(St_1) = \{C\}$, $In(St_2) = \{C, D, E\}$, $Out(St_2) = \{E\}$. We then construct the SDG $G_S = (V_S, E_S)$, with $V_S = In(St_1) \cup Out(St_1) \cup In(St_2) \cup Out(St_2) = \{A, B, C, D, E\}$. Furthermore, we

have edges $E_S = \{(A, C), (B, C), (C, E), (D, E), (E, E)\}$. The edges are annotated with the corresponding access function vectors $\phi_{St1,1}, \dots, \phi_{St2,3}$.

Note: While the “explicit” program CDAG $G = (V, E)$, where every vertex represents a single computation is indeed acyclic, the SDG $G_S = (V_S, E_S)$ may contain self-edges when a statement updates the loaded array ((E, E) in the example above). In G , one vertex corresponds to *one version* of a single array element, while in G_S , one vertex encapsulates *all versions* of all array elements.

5.5.4 SDG Subgraphs

Denote $I \subset V_S$ set of input vertices of G_S ($\forall A \in I : \text{indegree}(A) = 0$). Let $H \subset V_S \setminus I$ be a subset of the vertices of SDG $G_S = (V_S, E_S)$. The SDG subgraph $G_S[H]$ is a subgraph of G_S induced by the vertex set H . It corresponds to some subcomputation in which at least one vertex from each array in H was computed. We now use the analogous strategy to the X -Partitioning abstraction: since the optimal pebbling has an associated X -partition with certain properties (the dominator set constraint), we bound the cost of any pebbling by finding the maximum subcomputation among *all* valid X -partitions. We now show that every subcomputation in the optimal X -partition has a corresponding SDG subgraph $G_S[H]$. Therefore, finding $G_S[H_{opt}]$ that maximizes the computational intensity among *all* subgraphs bounds the size of the maximal subcomputation (which, in turn, bounds the I/O cost of any pebbling).

Recall that an optimal pebbling P has an associated X -partition $\mathcal{P}(X)$, where each $\mathcal{H} \in \mathcal{P}(X)$ represents a sequence of operations that are not interleaved with other subcomputations. Given G_S , each $\mathcal{H} \in \mathcal{P}(X)$ has an associated subgraph $G_S[H]$ s.t. every array vertex $A_i \in H$ represents an array from which at least one vertex was computed in \mathcal{H} .

Note that both the pebbling P and the partition $\mathcal{P}(X)$ depend on the size of the CDAG that is determined by the sizes of the iteration domains D^i . However, the SDG does not depend on them. Thus, by finding the subgraph that maximizes the computational intensity, we bound ρ for *any* combination of input parameters.

Definition 5.6. *The subgraph SOAP statement St_H of subgraph $G_S[H]$ is a single SOAP statement with the input $In(St_H) = \{A : A \notin H \wedge \exists B \in H : (A, B) \in E_S\}$. Additionally, for each vertex $B \in H$ that is not computed in H , that is $\nexists A \in H : (A, B) \in E_S$, self-edges $(B, B) \in E$ are preserved ($B \in In(St_H)$).*

Intuition. The subgraph statement St_H is a “virtual” SOAP statement that encapsulates multiple statements St_1, \dots, St_k . Given H , its subgraph statement’s inputs $In(St_H)$ are formed by merging inputs $\bigcup_{i=1}^k In(St_i) \setminus V(H)$ from all statements that form H , but are not in H . By the construction of the SDG, this is equivalent to the definition above: take all vertices $A \in V_s \setminus V(H)$ that have a child in $V(H)$, that is $\exists B \in V(H) : (A, B) \in E_S$ (see Figure 5.2).

This forms the lower bound on the number of inputs for a corresponding subcomputation \mathcal{H} : all the vertices from arrays $A_i \in V(H)$ could potentially be computed during \mathcal{H} and do not need to be loaded, but at least vertices from arrays $In(St_H)$ have to be accessed.

Example 5.8. Consider again the example from Figure 5.2. The set of input nodes is $I = \{A, B, D\}$. There are three possible subgraph statements: $H_1 = \{C\}$, with $In(St_{H_1}) = \{A, B\}$, $H_2 = \{C\}$ with $In(St_{H_2}) = \{C, D, E\}$, and $H_3 = \{C, E\}$ with $In(St_{H_3}) = \{A, B, D\}$. Note that by definition, the self-edge (C, C) is preserved in H_2 , but not in H_3 . Subgraphs H_1 and H_2 correspond to the input statements St_1 and St_2 . Subgraph H_3 encapsulates a subcomputation \mathcal{H} that computes some vertices from both arrays C and E , merging subcomputations St_1 and St_2 and reusing outputs from St_1 to compute E .

Then, we establish the following lemma:

Lemma 5.5. Given an X -partition $\mathcal{P}(X) = \{\mathcal{H}_1, \dots, \mathcal{H}_s\}$ of the k -statement SOAP, with its corresponding $G_S = (V_S, E_S)$, each subcomputation \mathcal{H} has an associated intensity $\rho_{\mathcal{H}} = \frac{|\mathcal{H}|}{|Dom_{min}(\mathcal{H})| - S}$ that is upper-bounded by the computational intensity of the subgraph statement St_H (Lemma 5.4).

Proof. Recall that given the subcomputation \mathcal{H} , its corresponding SDG subgraph H is constructed as follows: for each vertex $v \in V$ computed during \mathcal{H} belonging to some array A_i , add the corresponding array vertex s_i to H . Note that we allow a vertex recomputation: if some vertex is (re)computed during the optimal schedule of \mathcal{H} , its array vertex will belong to H .

Observe that by this construction and by definition of the subgraph statement, all arrays from which at least one vertex is loaded during \mathcal{H} are in $In(St_H)$. Furthermore, $In(St_H)$ is a subset of these arrays: during \mathcal{H} , there might be some loaded vertex from array $A_j \in H$, but, by definition of St_H , this array will not be in $In(St_H)$. Therefore, St_H lower bounds the input size of \mathcal{H} .

The last step of the proof is to observe that by Lemma 5.4, the computational intensity of St_H bounds the maximum number of computed vertices

for any $\mathcal{H}' \in \mathcal{P}(X)$ that belong to H , that is, the union of all arrays in H . But since all vertices that are computed in \mathcal{H} belong to one of these arrays, \mathcal{H} cannot have higher computational intensity. \square

5.5.5 SDG I/O Lower Bounds

We now proceed to establish a method to derive the I/O lower bounds of the multi-statement SOAP given its SDG $G_S = (V_S, E_S)$.

For each array vertex $A \in V_S$, denote $|A|$ as the total number of vertices in the CDAG that belong to array A . Denote further $\mathcal{S}(A)$ the set of all subgraphs of G_S that contain A . Then we prove the following theorem:

Theorem 5.1. *The I/O cost Q of a k -statement SOAP represented by the SDG $G_S = (V_S, E_S)$ is bounded by*

$$Q \geq \sum_{A \in V_S} \frac{|A|}{\max_{H \in \mathcal{S}(A)} \rho_H} \quad (5.10)$$

where $\max_{H \in \mathcal{S}(A)} \rho_H$ is the maximum computational intensity over all subgraph statements of subgraphs H that contain vertex A .

Proof. This theorem is a direct consequence of Lemma 5.5 and the fact that all vertices in CDAG G are associated with some array vertex in SDG G_S . Lemma 5.5, together with the definition of $\mathcal{S}(a)$, states that $\max_{H \in \mathcal{S}(a)} \rho_H$ is the upper bound on any subcomputation \mathcal{H} that contains any vertex from array a . Since there are $|a|$ vertices associated with a , at least $\frac{|a|}{\max_{H \in \mathcal{S}(a)} \rho_H}$ I/O operations must be performed to compute these vertices. Since the computational intensity expresses the average cost *per vertex*, even if some subcomputation in an optimal X -partition spans more than one array, this is already modeled by the set $\mathcal{S}(a)$. Therefore, we can sum the I/O costs per arrays a , yielding inequality 5.10. \square

Note that applying Theorem 5.1 requires iterating over all possible subgraphs. In the worst case, this yields exponential complexity, prohibiting scaling our method to large programs. However, many scientific applications contain a limited number of kernels with simple dependencies. In practice we observed that our approach scales well to programs containing up to 35 statements.

We evaluate our lower bound analysis on a wide range of applications, ranging from fundamental computational kernels and solvers to full workloads in hydrodynamics, numerical weather prediction, and deep learning.

The set of applications covers both the previously analyzed kernels (the Polybench suite [193], direct convolution), and kernels that were never analyzed before due to complicated dependency structures (multiple NN layers, diffusion, advection). Not only our tool covers broader class of programs than state-of-the-art approaches, but also it improves bounds generated by methods dedicated to specific narrower classes [162]. Improving I/O lower bounds has not only theoretical implications: loose bounds may not be applicable for generating corresponding parallel codes, as too many overapproximations may yield an invalid schedule.

In our experiments we use DaCe [69] to extract SOAP statements from Python and C code, and use MATLAB for symbolic analysis. ; see Appendix A for implementation details.

Polybench. As our first case study, we analyze Polybench [193], a polyhedral application benchmark suite composed of 30 programs from several domains, including linear algebra kernels, linear solvers, data mining, and computational biology. Prior best results were obtained by IOLB [162], a tool specifically designed for analyzing I/O lower bounds of affine programs. We summarize the results in Table 5.2, listing the leading order term for brevity.

We find that SOAP analysis derives tight I/O lower bounds for all Polybench kernels. Analyzing these programs as multi-statement SOAP either reproduces existing tight bounds, or improves them by constant factors (e.g., in Cholesky decomposition) on 14 out of 30 applications (Table 5.2). Of particular note is *adi* (Alternating Direction Implicit solver). Our algorithm detected a possible tiling in the time dimension, yielding the lower bound $(12N^2T)/\sqrt{5}$, compared to N^2T reported by Olivry et al. [162]. However, due to dependency chains incurred by alternating directions, such tiling may violate loop-carried dependency constraints, which our algorithm relaxes. A parallel machine could potentially take advantage of this tiling scheme, possibly providing super-linear communication reduction. However, this is outside of the scope of this paper.

Neural Networks. Analyzing I/O lower bounds of neural networks is a nascent field, and so far only single-layer convolution was analyzed [59, 192]. We improve the previously-reported bound reported by Zhang et al. [192] by a factor of 8.

	Kernel	SOAP I/O Bound	Improvement over state-of-the-art
Polybench [162]	adi	$\frac{12N^2T}{\sqrt{5}}$	$\frac{12}{\sqrt{5}}$
	atax	MN	1
	bicg	MN	1
	cholesky	$\frac{N^3}{3\sqrt{5}}$	2
	correlation	$\frac{M^2N}{\sqrt{5}}$	2
	covariance	$\frac{M^2N}{\sqrt{5}}$	2
	deriche	$3HW$	3
	doitgen	$\frac{2N_P^2N_QN_R}{\sqrt{5}}$	1
	durbin	$\frac{3N^2}{2}$	3
	fdtd2d	$\frac{2\sqrt{3}N_XN_YT}{\sqrt{5}}$	$6\sqrt{6}$
	floyd-warshall	$\frac{2N^3}{\sqrt{5}}$	2
	gemm	$\frac{2N^2}{\sqrt{5}}$	1
	gemver	N^2	1
	gesummv	$2N^2$	1
	gramschmidt	$\frac{MN^2}{\sqrt{5}}$	1
	heat3d	$\frac{6N^3T}{\sqrt[3]{5}}$	$\frac{32}{3\sqrt[3]{5}}$
	jacobi1d	$\frac{2NT}{5}$	8
	jacobi2d	$\frac{4N^2T}{\sqrt{5}}$	$6\sqrt{3}$
	2mm	$\frac{4N^3}{\sqrt{5}}$	1
	3mm	$\frac{6N^3}{\sqrt{5}}$	1
	lu	$\frac{2N^3}{3\sqrt{5}}$	1
	ludcmp	$\frac{2N^3}{3\sqrt{5}}$	1
	mvt	N^2	1
	nussinov	$\frac{N^3}{3\sqrt{5}}$	2
	seidel2d	$\frac{4N^2T}{\sqrt{5}}$	$6\sqrt{3}$
	symm	$\frac{2M^2N}{\sqrt{5}}$	1
	syr2k	$\frac{2MN^2}{\sqrt{5}}$	2
	syrk	$\frac{MN^2}{\sqrt{5}}$	2
trisolv	$\frac{N^2}{2}$	1	
trmm	$\frac{M^2N}{\sqrt{5}}$	1	
Neural Networks	Direct conv.	$\frac{2C_{in}C_{out}H_{out}NW_{out}W_{ker}H_{ker}}{\sqrt{5}}$	8
	Softmax	$4BHMN$	—
	MLP	$\frac{2N(fc_1fc_2+fc_1inp+fc_2out)}{\sqrt{5}}$	—
	LeNet-5	$\frac{300\sqrt{2}CHNW}{\sqrt{5}}$	—
	BERT Encoder	$\frac{4BHP(L+2HP)}{\sqrt{5}}$	—
Various	LULESH	$22 \cdot \text{numElem}$	—
	horizontal diff.	$2IJK$	—
	vertical adv.	$5IJK$	—

Table 5.2: Simplified leading-order terms of the I/O lower bounds extracted from multi-statement SOAP and previous state-of-the-art. For the direct convolution, the best previous leading-order bound was published by Zhang

5.5.6 *New Lower Bounds*

Analyzing SOAP and the SDG representation enables capturing complex data dependencies in programs with a large number of statements. To demonstrate this, we study larger programs in three fields, where no previous I/O bounds are known. If an application contains both SOAP and data-dependent kernels, we find a SOAP representation that bounds the access sizes from below.

LULESH. The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) [194] application is an unstructured phy-sics simulation. We analyze the main computational kernel, totaling over 60% of runtime within one time-step of the simulation from the full C++ source code. As LULESH falls outside the purview of affine programs, this result is the first reported I/O lower bound.

Numerical Weather Prediction. We select two benchmark stencil applications from the COSMO Weather Model [195] — horizontal diffusion and vertical advection — representatives of the two major workload types in the model’s dynamical core.

Deep Neural Networks. For deep learning, we choose both individual representative operators (Convolution and Softmax) and network-scale benchmarks. Previous approaches only study data movement empirically [196]. To the best of our knowledge, we are the first to obtain I/O lower bounds for full networks, including a Multi-Layer Perceptron (MLP), the LeNet-5 CNN [197], and a BERT Transformer encoder [198].

5.6 RELATED WORK

I/O analysis spans almost the entire history of general-purpose computer architectures, and graph pebbling abstractions were among the first methods to model memory requirements. Dating back to challenges with the register allocation problem [128], pebbles were also used to prove space-time tradeoffs [21] and maximum parallel speedups by investigating circuit depths [199]. Arguably the most influential pebbling abstraction work is the red-blue pebble game by Hong and Kung [22] that explicitly models load and store operations in a two-level-deep memory hierarchy. This work was extended numerous times, by: adding blocked access [23], multiple memory hierarchies [200], or introducing additional pebbles to allow CDAG compositions [24]. Demaine and Liu proved that finding the optimal pebbling in a standard and no-deletion red-blue pebble game is PSPACE-complete [191].

Papp and Wattenhofer introduced a game variant with a non-zero computation cost and investigated pebbling approximation algorithms [201].

Although the importance of data movement minimization is beyond doubt, the general solution for arbitrary algorithms is still an open problem. Therefore, many works were dedicated to investigate lower bounds only for single algorithms (often with accompanying implementations), like matrix-matrix multiplication [52, 64, 65, 202], LU [52] and Cholesky decompositions [53, 155]. Ballard et al. [54] present an extensive collection of linear algebra algorithms. Moreover, a large body of work exists for minimizing communication in irregular algorithms [203, 204], such as Betweenness Centrality [113], min cuts [205], BFS [112], matchings [206], vertex similarity coefficients [207], or general graph computations [208, 209]. Many of them use linear algebra based formulations [110]. Recently, convolution networks gained high attention. The first asymptotic I/O lower bound for single-layer direct convolution was proved by Demmel et al. [59]. Chen et al. [210] propose a matching implementation, and Zhang et al. [192] present the first non-asymptotic I/O lower bound for Winograd convolution.

In parallel with the development of I/O minimizing implementations for particular algorithms, several works investigated I/O lower bounds for whole classes of programs. Christ et al. [57] use a discrete version of Loomis-Whitney inequality to derive asymptotic lower bounds for single-statement programs nested in affine loops. Demmel and Rusciano [58] extended this work and use discrete Hölder-Brascamp-Lieb inequalities to find optimal tilings for such programs. The polyhedral model [161] is widely used in practice by many compilers [211, 212]. However, polyhedral methods have their own limitations: 1) they cannot capture non-affine loops [213]; 2) while the representation of a program is polynomial, finding optimal transformations is still NP-hard [138]; 3) they are inapplicable for many neural network architectures, e.g., the Winograd algorithm for convolution [192].

Recently, Olivry et al. [162] presented IOLB — a tool for automatic derivation of non-parametric I/O lower bounds for programs that can be modeled by the polyhedral framework. IOLB employs both “geometric” projection-based bounds based on the HBL inequality [57], as well as the wavefront-based approach from Elango [214]. To the best of our knowledge, this is the only method that can handle multiple-statement programs. However, the IOLB model explicitly disallows recomputation that may be used to decrease the I/O cost, e.g., in the Winograd convolution algorithm, back-propagation, or vertical advection. Furthermore, the framework is strictly

limited to affine access programs. Even then, our method is able to improve those bounds by up to a factor of $6\sqrt{6}$ (fdd2d) using a single, general method without the need to use application-specific techniques, such as wavefront-based reasoning.

5.7 SUMMARY

In this chapter we introduce SOAP — a broad class of statically analyzable programs. Using the explicit assumptions on the allowed overlap between arrays, we are able to precisely count the number of accessed vertices on the induced parametric CDAG. This stands in contrast with many state-of-the-art approaches that are based on bounding projection sizes, as they need to underapproximate their union size, often resulting in a significant slack in constant factors of their bounds. Our single method is able to reproduce or improve existing lower bounds for many important scientific kernels from various domains, ranging from $2\times$ increase in the lower bound for linear algebra (cholesky, syrk), to more than $10\times$ for stencil applications (fdd2d, heat3d).

Our SDG abstraction precisely models data dependencies in multiple-statement programs. It directly captures input and output reuse, and allows data recomputation. Armed with these tools, we are the first to establish I/O lower bounds for entire neural networks, as well as core components of the popular Transformer architecture.

We believe that our work will be further extended to handle data-dependent accesses (e.g., sparse matrices), as well as scale better with input program size. The derived maximum subcomputation sizes can guide compiler optimizations and development of new communication-optimal algorithms through tiling, parallelization, or loop fusion transformations.

CONCLUSIONS AND FUTURE WORK

In this dissertation, we improve deriving fundamental performance metrics of scientific applications, such as arithmetic, parallel, and I/O complexities. Our main goal is obtaining tight bounds, thus addressing the problem of hidden constants that may impact actual performance. The proofs of obtained bounds are often constructive, providing optimal or close to optimal algorithms. We also put a significant effort into fine-tuning the derived algorithms to utilize modern distributed machines equipped with deep memory hierarchies and accelerators.

To achieve this goal, we first introduce new models of iteration spaces and memory accesses within them. The precise count of both the number of iterations and array references allows us to derive arithmetic and I/O complexities and parallel efficiency for input programs written in C, Fortran, and Python. The insights allow us to reason about the scalability of input algorithms and obtain new algorithms that minimize both the I/O movement across the memory hierarchy and between parallel processors. Finally, we implement the derived I/O minimizing matrix multiplication and factorizations algorithms using a series of optimizations, such as the processor grid optimization, local densifications, latency-minimizing tournament pivoting, and layout transformations. The resulting open-source libraries outperform all compared state-of-the-art solutions, confirming not only the theoretical contribution of the models but also providing practical applications in the algorithmic design.

We start by presenting a method to symbolically count loop iterations, even in the presence of loop updates that cannot be captured by the Presburger arithmetic (Chapter 2). The obtained solution is either exact, or both the upper and the lower bounds are provided by our bounded sum approximation algorithm. In the presence of data-dependent control flow, corresponding loop iterations are parametrized, allowing the user to provide appropriate expected upper and lower bounds. For explicitly parallel programs, the parallel efficiency, as well as the work-depth analysis, is provided.

We then focus our work on the data movement minimization. We use the red-blue pebble game to establish the X-Partitioning abstraction and prove Lemma 3.3, which explicitly captures data reuse between subcom-

putations of the input CDAGs (Chapter 3). We then use it to analyze the classical MMM algorithm and establish tight sequential (Theorem 3.1) and parallel (Theorem 3.2) I/O lower bounds. The proofs are constructive: the resulting COSMA algorithm is I/O optimal up to the factor of $\frac{\sqrt{S}}{\sqrt{S+1}-1}$ for all combinations of matrix dimensions, processor counts, and memory sizes. Implemented optimizations, both algorithmic and hardware-oriented, secure the best time-to-solution performance compared to the state-of-the-art libraries in all evaluated benchmarks, providing up to 12.8x speedup. Moreover, the “bottom-up” parallelization strategy can be generalized to other algorithms that can operate on non-cubic domains.

We extend the results from Chapter 3, both theoretical and practical, to cover a wide class of programs: DAAP. Within this class, we show how we can apply Theorem 3.1 by precisely counting the number of array accesses (Lemma 4.3), exploiting the disjoint access property. We use our DAAP model to LU and Cholesky factorizations and obtain new parallel I/O lower bounds, together with corresponding communication-avoiding schedules. The resulting algorithms — *CONfLUX* and *CONfCHOX*— not only communicate less than asymptotically optimal state-of-the-art 3D algorithms by a factor of up to 1.6x but also outperform all evaluated libraries by up to 3 times.

We then further generalize the DAAP model to model overlapping array accesses: a common pattern in many scientific applications. The introduced SDG abstraction allows for precise data movement analysis in large programs consisting of multiple computation kernels. We use SOAP to improve existing lower bounds for a series of important compute kernels, covering the entire Polybench suite and extending beyond the polyhedral model. With our method, we tighten I/O lower bounds for 14 Polybench kernels (up to 14 times) and we establish first I/O lower bounds for entire neural networks, such as LeNet-5. The method provides powerful compiler hints on the tiling, parallelization, and loop fusion transformations. Combined with the precise loop counting algorithm from Chapter 2 and optimizations developed in Chapters 3 and 4, such as processor grid optimizations, “bottom-up” parallel decomposition, and mixed 1D/2.5D parallelization, our work can serve to automatically generate more communication-avoiding high-performance implementations.

6.1 FUTURE WORK

The path to optimality is open-ended: as discussed in Chapter 1, the notion of optimality intrinsically depends on both the definition and the adopted computation model. We plan to pursue several projects in the future that can be categorized into three paths: theoretical analysis, algorithmic design, and compiler automation.

Theoretical analysis. While the SOAP model proves to be (a) general: it extends beyond the polyhedral analysis; (b) precise: it improves existing lower bounds; and (c) constructive: it provides direct information on optimal kernel fusion and tiling, it can still be further extended. One of the main directions we wish to continue our research is modeling sparse data structures — notoriously hard objects to precisely model, especially regarding the access pattern. While there is significant progress in designing communication-avoiding, high-performance sparse linear algebra algorithms [215–217] we wish to explore further possibilities of attaining tight (presumably, structure-dependent) communication lower bounds, not only for sparse matrix-matrix multiplication but for general linear algebra algorithms. General graph computations, such as graph mining or graph neural networks, are also promising candidates for I/O complexity analysis.

Algorithmic design. As shown in Chapter 4, complex data flow, such as pivoting or loop-carried dependencies in the reduction dimension, prohibits straightforward parallelization of sequential schedules. Furthermore, communication optimality alone does not guarantee the best performance in practice. We aim to continue implementing our algorithms on modern hardware, such as GPUs and FPGAs, similarly to our work on porting COSMA to HLS (High-Level Synthesis) [218]. Communication-avoiding parallel decompositions, combined with intra-node optimizations introduced in Chapters 3 and 4, can also be used in general tensor operations. Combined with possibly sparse data structures, achieving peak performance, similar to dense MMM, is a challenging task worth pursuing.

Compiler automation. We implement our SOAP analysis in Python using the DaCe data-centric framework [69]. DaCe is a powerful tool that compiles input codes to the data-centric intermediate representation, called SDFG (Stateful DataFlow Graph). This representation allows performing a multitude of dataflow-oriented optimizations, such as automatic parallelization and tiling. It comes with multiple back-ends to target CPUs, GPUs, and FPGAs.

While the transformations can be applied automatically, their parameters (e.g., tile sizes) have to be either provided by the user or inferred by expensive (and possibly, suboptimal) autotuning. SOAP can be integrated with DaCe to provide explicit data movement parametric cost and provide DaCe with transformation parameters, offering a tool that can automatically generate a provably data movement optimal, parallel, high-performance code for all modern hardware platforms.

BIBLIOGRAPHY

1. Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society* 2, 230 (1937) (cit. on pp. 1, 15).
2. Hartmanis, J. & Stearns, R. E. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 285 (1965) (cit. on p. 1).
3. Kahale, N., Leighton, T., Ma, Y., Plaxton, C. G., Suel, T. & Szemerédi, E. Lower bounds for sorting networks in *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing* (1995), 437 (cit. on p. 1).
4. Plaxton, C. G. & Suel, T. A lower bound for sorting networks based on the shuffle permutation. *Mathematical Systems Theory*, 491 (1994) (cit. on p. 1).
5. Batcher, K. E. *Sorting networks and their applications* in *Proceedings of the April 30–May 2, 1968, spring joint computer conference* (1968), 307 (cit. on p. 1).
6. Bshouty, N. H. A lower bound for matrix multiplication. *SIAM Journal on Computing* 18, 759 (1989) (cit. on p. 1).
7. Strassen, V. Gaussian Elimination is Not Optimal. *Numer. Math.*, 354 (1969) (cit. on pp. 1, 35).
8. Alman, J. & Williams, V. V. *A refined laser method and faster matrix multiplication* in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2021), 522 (cit. on p. 1).
9. Ranković, V., Kos, A., Tomažič, S. & Milutinović, V. *Performance of the bitonic mergesort network on a dataflow computer* in *2013 21st Telecommunications Forum Telfor (TELFOR)* (2013), 849 (cit. on p. 2).
10. Mu, Q., Cui, L. & Song, Y. *The implementation and optimization of Bitonic sort algorithm based on CUDA* in (2015) (cit. on p. 2).
11. Ballard, G., Demmel, J., Holtz, O., Lipshitz, B. & Schwartz, O. *Communication-optimal parallel algorithm for strassen’s matrix multiplication* in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures* (2012), 193 (cit. on pp. 2, 6, 35, 40).

12. Ballard, G., Demmel, J., Holtz, O. & Schwartz, O. Graph expansion and communication costs of fast matrix multiplication. *JACM* (2012) (cit. on pp. 2, 79).
13. Zhang, P. & Gao, Y. *Matrix multiplication on high-density multi-GPU architectures: theoretical and experimental investigations* in *International Conference on High Performance Computing* (2015), 17 (cit. on pp. 2, 35).
14. Papadimitriou, C. H. *Complexity theory. Reading: Addison Wesley* (1994) (cit. on p. 2).
15. Harvey, D. & Van Der Hoeven, J. Integer multiplication in time $O(n \log n)$. *Annals of Mathematics* **193**, 563 (2021) (cit. on p. 2).
16. Günther, S. M., Appel, N. & Carle, G. Galois Field Arithmetics for Linear Network Coding using AVX512 Instruction Set Extensions. *arXiv preprint arXiv:1909.02871* (2019) (cit. on p. 3).
17. Von Neumann, J. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 27 (1993) (cit. on p. 3).
18. De Fine Licht, J., Besta, M., Meierhans, S. & Hoefler, T. *Transformations of high-level synthesis codes for high-performance computing* in (IEEE, 2020), 1014 (cit. on p. 3).
19. Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Commun. ACM*, 613. <https://doi.org/10.1145/359576.359579> (1978) (cit. on p. 3).
20. Gilbert, J. R., Lengauer, T. & Tarjan, R. E. The pebbling problem is complete in polynomial space. *SIAM Journal on Computing* **9**, 513 (1980) (cit. on pp. 3, 39).
21. Paul, W. J. & Tarjan, R. E. Time-space trade-offs in a pebble game. *Acta Informatica*, 111 (1978) (cit. on pp. 3, 146).
22. Hong, J. & Kung, H. *I/O complexity: The red-blue pebble game* in *STOC* (1981), 326 (cit. on pp. 3, 7, 8, 38, 39, 43, 45–47, 49, 78, 81, 86, 87, 118, 121, 146).
23. Aggarwal, A. & Vitter Jeffrey, S. The Input/Output Complexity of Sorting and Related Problems. *CACM* (1988) (cit. on pp. 3, 5, 37, 78, 79, 118, 146).
24. Elango, V., Rastello, F., Pouchet, L.-N., Ramanujam, J. & Sadayappan, P. *Data access complexity: The red/blue pebble game revisited* tech. rep. (Technical Report, 2013) (cit. on pp. 3, 78, 118, 121, 146).

25. Kwasniewski, G., Ben-Nun, T., Gianinazzi, L., Calotoiu, A., Schneider, T., Ziogas, A. N., Besta, M. & Hoefler, T. *Pebbles, Graphs, and a Pinch of Combinatorics: Towards Tight I/O Lower Bounds for Statically Analyzable Programs* in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Association for Computing Machinery, Virtual Event, USA, 2021), 328. <https://doi.org/10.1145/3409964.3461796> (cit. on pp. 3, 6, 121).
26. Dennard, R. H., Gaensslen, F. H., Yu, H.-N., Rideout, V. L., Bassous, E. & LeBlanc, A. R. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 256 (1974) (cit. on pp. 3, 81).
27. Mamaluy, D. & Gao, X. The fundamental downscaling limit of field effect transistors. *Applied Physics Letters*, 193503 (2015) (cit. on p. 4).
28. Martin, C. Post-dennard scaling and the final years of Moores Law. *Technical report* (2014) (cit. on p. 4).
29. Dijkstra, E. W. *A note on two problems in connexion with graphs* in (1959), 269 (cit. on p. 4).
30. Bellman, R. On a routing problem. *Quarterly of applied mathematics* **16**, 87 (1958) (cit. on p. 4).
31. Brent, R. P. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)*, 201 (1974) (cit. on pp. 4, 12).
32. Valiant, L. G. A bridging model for parallel computation. *Communications of the ACM* **33**, 103 (1990) (cit. on p. 4).
33. Grama, A. Y., Gupta, A. & Kumar, V. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel & Distributed Technology: Systems & Applications*, 12 (1993) (cit. on p. 4).
34. Vaibhaw, V. *MMM performance engineering* <https://github.com/vaibhawvipul/performance-engineering>. 2021 (cit. on p. 5).
35. Wahib, M. & Maruyama, N. *Scalable kernel fusion for memory-bound GPU applications* in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), 191 (cit. on p. 5).
36. Snir, M. Depth-size trade-offs for parallel prefix computation. *Journal of Algorithms*, 185 (1986) (cit. on p. 5).

37. Solomonik, E., Carson, E., Knight, N. & Demmel, J. *Trade-offs between synchronization, communication, and computation in parallel linear algebra computations* in (ACM New York, NY, USA, 2017), 1 (cit. on pp. 5, 6, 78, 81, 121).
38. Fuhrer, O., Chadha, T., Hoefler, T., Kwasniewski, G., Lapillonne, X., Leutwyler, D., Lüthi, D., Osuna, C., Schär, C., Schulthess, T. C., *et al.* Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0. *Geoscientific Model Development*, 1665 (2018) (cit. on p. 5).
39. McCalpin, J. D. *et al.* Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 2 (1995) (cit. on p. 5).
40. Del Ben, M., Schütt, O., Wentz, T., Messmer, P., Hutter, J. & Vandevondele, J. Enabling simulation at the fifth rung of DFT: Large scale RPA calculations with excellent time to solution. *Computer Physics Communications* 187, 120 (2015) (cit. on pp. 6, 35, 70, 81, 108, 116, 121).
41. Srebro, N. *Learning with matrix factorizations* in (2004) (cit. on p. 6).
42. Solomonik, E., Matthews, D., Hammond, J. & Demmel, J. *Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions* in 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (2013), 813 (cit. on pp. 6, 37).
43. Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A. & Bastoul, C. *The polyhedral model is more widely applicable than you think* in *International Conference on Compiler Construction* (2010), 283 (cit. on pp. 6, 15, 33, 117, 118).
44. Yuster, R. & Zwick, U. Fast sparse matrix multiplication. *ACM Transactions On Algorithms (TALG)* 1, 2 (2005) (cit. on p. 6).
45. Saad, Y. ILUT: A dual threshold incomplete LU factorization. *Numerical linear algebra with applications*, 387 (1994) (cit. on p. 6).
46. Buttari, A., Dongarra, J., Langou, J., Langou, J., Luszczek, P. & Kurzak, J. Mixed precision iterative refinement techniques for the solution of dense linear systems. *The International Journal of High Performance Computing Applications*, 457 (2007) (cit. on p. 6).

47. Haidar, A., Tomov, S., Dongarra, J. & Higham, N. J. *Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (2018), 47 (cit. on pp. 6, 74, 118).
48. Wolfe, M. *More iteration space tiling* in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing* (1989), 655 (cit. on pp. 6, 60, 79).
49. Dinh, G. & Demmel, J. *Communication-Optimal Tilings for Projective Nested Loops with Arbitrary Bounds*. *arXiv preprint arXiv:2003.00119* (2020) (cit. on pp. 6, 85, 118, 122).
50. Mehta, S., Lin, P.-H. & Yew, P.-C. *Revisiting loop fusion in the polyhedral framework* in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2014), 233 (cit. on pp. 6, 117, 118).
51. Irony, D., Toledo, S. & Tiskin, A. *Communication lower bounds for distributed-memory matrix multiplication*. *Journal of Parallel and Distributed Computing*, 1017 (2004) (cit. on pp. 6, 7, 37, 49, 78, 79, 81).
52. Solomonik, E. & Demmel, J. *Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms* in *Euro-Par 2011 Parallel Processing* (eds Jeannot, E., Namyst, R. & Roman, J.) (Springer Berlin Heidelberg, 2011), 90. http://dx.doi.org/10.1007/978-3-642-23397-5%5C_10 (cit. on pp. 6, 7, 36, 37, 40, 62, 63, 79, 81–83, 98, 99, 101, 107, 108, 147).
53. Ballard, G., Demmel, J., Holtz, O. & Schwartz, O. *Communication-optimal parallel and sequential Cholesky decomposition*. *SIAM Journal on Scientific Computing*, 3495 (2010) (cit. on pp. 6, 81, 147).
54. Ballard, G., Carson, E., Demmel, J., Hoemmen, M., Knight, N. & Schwartz, O. *Communication lower bounds and optimal algorithms for numerical linear algebra*. *Acta Numerica*, 1 (2014) (cit. on pp. 6, 147).
55. Ballard, G., Demmel, J., Holtz, O. & Schwartz, O. *Minimizing communication in numerical linear algebra*. *SIAM Journal on Matrix Analysis and Applications*, 866 (2011) (cit. on pp. 6, 117, 118).
56. Smith, T. M. & van de Geijn, R. A. *Pushing the bounds for matrix-matrix multiplication*. *CoRR abs/1702.02017* (2017) (cit. on pp. 6, 7, 49, 58, 78).

57. Christ, M., Demmel, J., Knight, N., Scanlon, T. & Yelick, K. Communication lower bounds and optimal algorithms for programs that reference arrays—Part 1. *arXiv preprint arXiv:1308.0068* (2013) (cit. on pp. 6, 81, 91, 118, 121, 122, 147).
58. Demmel, J. & Rusciano, A. Parallelepipeds obtaining HBL lower bounds. *arXiv preprint arXiv:1611.05944* (2016) (cit. on pp. 6, 118, 122, 147).
59. Demmel, J. & Dinh, G. Communication-optimal convolutional neural nets. *arXiv preprint arXiv:1802.06905* (2018) (cit. on pp. 6, 118, 122, 139, 144, 147).
60. Cannon, L. E. *A Cellular Computer to Implement the Kalman Filter Algorithm* PhD thesis (1969) (cit. on pp. 7, 35, 40, 62, 79).
61. Choi, J., Walker, D. W. & Dongarra, J. J. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 543 (1994) (cit. on pp. 7, 35).
62. Van De Geijn, R. A. & Watts, J. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* **9**, 255 (1997) (cit. on pp. 7, 35, 37, 38, 63).
63. Agarwal, R. C., Balle, S. M., Gustavson, F. G., Joshi, M. & Palkar, P. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* **39**, 575 (1995) (cit. on pp. 7, 35, 79).
64. Demmel, J., Eliahu, D., Fox, A., Kamil, S., Lipshitz, B., Schwartz, O. & Spillinger, O. *Communication-optimal parallel recursive rectangular matrix multiplication in 2013 IEEE 27th International Symposium on Parallel and Distributed Processing* (2013), 261 (cit. on pp. 7, 36, 37, 62, 63, 69, 79, 147).
65. Kwasniewski, G., Kabić, M., Besta, M., VandeVondele, J., Solcà, R. & Hoefler, T. *Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)* (2019) (cit. on pp. 7, 35, 118, 121, 147).
66. Grosser, T., Verdoolaege, S. & Cohen, A. *Polyhedral AST generation is more than scanning polyhedra in (ACM New York, NY, USA, 2015)*, 1 (cit. on p. 8).

67. Lattner, C. & Adve, V. *LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation* in (San Jose, CA, USA, 2004), 75 (cit. on p. 8).
68. Kühne, T. D., Iannuzzi, M., Del Ben, M., Rybkin, V. V., Seewald, P., Stein, F., Laino, T., Khaliullin, R. Z., Schütt, O., Schiffmann, F., *et al.* CP2K: An electronic structure and molecular dynamics software package-Quickstep: Efficient and accurate electronic structure calculations. *The Journal of Chemical Physics*, 194103 (2020) (cit. on pp. 9, 81).
69. Ben-Nun, T., de Fine Licht, J., Ziogas, A. N., Schneider, T. & Hoefler, T. *Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2019) (cit. on pp. 9, 144, 151).
70. Goodale, T., Allen, G., Lanfermann, G., Massó, J., Radke, T., Seidel, E. & Shalf, J. *The cactus framework and toolkit: Design and applications in High Performance Computing for Computational Science* (2003) (cit. on p. 11).
71. DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., *et al.* *Liszt: a domain specific language for building portable mesh-based PDE solvers* in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis* (2011), 1 (cit. on p. 11).
72. Shiloach, Y. & Vishkin, U. An $O(n^2 \log n)$ parallel max-flow algorithm. *Journal of Algorithms* 3, 128 (1982) (cit. on p. 12).
73. JáJá, J. *An Introduction to Parallel Algorithms* (1992) (cit. on p. 12).
74. Karp, R. M., Ramachandran, V. & van Leeuwen, J. Handbook of theoretical computer science. *chapter Parallel Algorithms for Shared-Memory Machines*, 869 (1990) (cit. on p. 12).
75. Amdahl, G. M. *Validity of the single processor approach to achieving large scale computing capabilities* in *Proceedings of the April 18-20, 1967, spring joint computer conference* (1967), 483 (cit. on p. 12).
76. Dahl, O.-J., Dijkstra, E. W. & Hoare, C. A. R. *Structured programming* (Academic Press Ltd., 1972) (cit. on pp. 12, 16).
77. Knuth, D. *Sorting and Searching: The Art of Computer Programming* (1973) (cit. on p. 14).

78. Barvinok, A. I. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research* **19**, 769 (1994) (cit. on pp. 15, 33).
79. Bernardin, L. A review of symbolic solvers. *ACM SIGSAM Bulletin* **30**, 9 (1996) (cit. on p. 20).
80. Leiserson, C. E., Rivest, R. L., Stein, C. & Cormen, T. H. *Introduction to Algorithms* (2001) (cit. on p. 22).
81. Lattner, C. & Adve, V. *LLVM: A compilation framework for lifelong program analysis & transformation in International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), 75 (cit. on p. 27).
82. Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., et al. *The NAS parallel benchmarks summary and preliminary results in Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing* (1991), 158 (cit. on p. 28).
83. Barrett, R. F., Heroux, M. A., Lin, P. T., Vaughan, C. T. & Williams, A. B. in *Proceedings of the 2011 companion on High Performance Computing Networking, Storage and Analysis Companion* **1** (2011) (cit. on p. 28).
84. Blelloch, G. E. Programming parallel algorithms. *Communications of the ACM* **39**, 85 (1996) (cit. on p. 32).
85. Rodriguez-Carbonell, E. & Kapur, D. *Automatic generation of polynomial loop invariants: Algebraic foundations in Proceedings of the 2004 international symposium on Symbolic and algebraic computation* (2004), 266 (cit. on p. 33).
86. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P. & Nori, A. V. *A data driven approach for algebraic loop invariants in European Symposium on Programming* (2013), 574 (cit. on p. 33).
87. Matringe, N., Moura, A. V. & Rebiha, R. *Generating invariants for non-linear hybrid systems by linear algebraic methods in International Static Analysis Symposium* (2010), 373 (cit. on p. 33).
88. Zuleger, F., Gulwani, S., Sinn, M. & Veith, H. *Bound analysis of imperative programs with the size-change abstraction in International Static Analysis Symposium* (2011), 280 (cit. on p. 33).
89. Ben-Amram, A. M. & Genaim, S. On the linear ranking problem for integer linear-constraint loops. *ACM SIGPLAN Notices* **48**, 51 (2013) (cit. on p. 33).

90. Alias, C., Darte, A., Feautrier, P. & Gonnord, L. *Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs* in *International Static Analysis Symposium* (2010), 117 (cit. on p. 33).
91. Blanc, R., Henzinger, T. A., Hottelier, T. & Kovács, L. *ABC: algebraic bound computation for loops* in *International Conference on Logic for Programming Artificial Intelligence and Reasoning* (2010), 103 (cit. on p. 33).
92. Barnes, B. J., Rountree, B., Lowenthal, D. K., Reeves, J., De Supinski, B. & Schulz, M. *A regression-based approach to scalability prediction* in *Proceedings of the 22nd annual international conference on Supercomputing* (2008), 368 (cit. on p. 33).
93. Calotoiu, A., Hoefler, T., Poke, M. & Wolf, F. *Using automated performance modeling to find scalability bugs in complex codes* in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013), 1 (cit. on p. 33).
94. Ipek, E., De Supinski, B. R., Schulz, M. & McKee, S. A. *An approach to performance prediction for parallel applications* in *European Conference on Parallel Processing* (2005), 196 (cit. on p. 33).
95. Lee, B. C., Brooks, D. M., de Supinski, B. R., Schulz, M., Singh, K. & McKee, S. A. *Methods of inference and learning for performance modeling of parallel applications* in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2007), 249 (cit. on p. 33).
96. Carrington, L., Snively, A., Gao, X. & Wolter, N. *A performance prediction framework for scientific applications* in *International Conference on Computational Science* (2003), 926 (cit. on p. 33).
97. Marin, G. & Mellor-Crummey, J. *Cross-architecture performance predictions for scientific applications using parameterized models* in *Proceedings of the joint international conference on Measurement and modeling of computer systems* (2004), 2 (cit. on p. 33).
98. Zhai, J., Chen, W. & Zheng, W. *Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node*. *ACM sigplan notices* **45**, 305 (2010) (cit. on p. 33).
99. Yang, L. T., Ma, X. & Mueller, F. *Cross-platform performance prediction of parallel applications using partial execution* in *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing* (2005), 40 (cit. on p. 33).

100. Jain, R. *The Art Of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling* (2008) (cit. on p. 33).
101. Hoefler, T., Gropp, W., Snir, M. & Kramer, W. *Performance Modeling for Systematic Performance Tuning in Proc. of the 2011 ACM/IEEE Supercomputing* (2011) (cit. on p. 33).
102. Kerbyson, D. J., Alme, H. J., Hoisie, A., Petrini, F., Wasserman, H. J. & Gittings, M. *Predictive performance and scalability modeling of a large-scale application in Proceedings of the 2001 ACM/IEEE conference on Supercomputing* (2001), 37 (cit. on p. 33).
103. Bauer, G., Gottlieb, S. & Hoefler, T. *Performance modeling and comparative analysis of the MILC lattice QCD application su3_rmd in 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)* (2012), 652 (cit. on p. 33).
104. Meyer, C. D. *Matrix analysis and applied linear algebra* (SIAM, 2000) (cit. on pp. 35, 81, 121).
105. Chatelin, F. *Eigenvalues of Matrices: Revised Edition* (Siam, 2012) (cit. on p. 35).
106. Choi, J., Dongarra, J. J., Ostrouchov, L. S., Petitet, A. P., Walker, D. W. & Whaley, R. C. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming* **5**, 173 (1996) (cit. on p. 35).
107. Ben-Nun, T. & Hoefler, T. *Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis in* (2018) (cit. on p. 35).
108. Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. *Introduction to algorithms* (MIT press, 2009) (cit. on p. 35).
109. Azad, A., Buluç, A. & Gilbert, J. *Parallel triangle counting and enumeration using matrix algebra in 2015 IEEE International Parallel and Distributed Processing Symposium Workshop* (2015), 804 (cit. on p. 35).
110. Kepner, J., Aaltonen, P., Bader, D., Buluç, A., Franchetti, F., Gilbert, J., Hutchison, D., Kumar, M., Lumsdaine, A., Meyerhenke, H., *et al.* *Mathematical foundations of the GraphBLAS in 2016 IEEE High Performance Extreme Computing Conference (HPEC)* (2016), 1 (cit. on pp. 35, 147).
111. Ng, A. Y., Jordan, M. I. & Weiss, Y. *On spectral clustering: Analysis and an algorithm in Advances in neural information processing systems* (2002), 849 (cit. on p. 35).

112. Besta, M., Marending, F., Solomonik, E. & Hoefler, T. *SlimSell: A Vectorizable Graph Representation for Breadth-First Search in IPDPS* (2017) (cit. on pp. 35, 147).
113. Solomonik, E., Besta, M., Vella, F. & Hoefler, T. *Scaling betweenness centrality using communication-efficient sparse matrix multiplication in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), 1 (cit. on pp. 35, 81, 121, 147).
114. Li, J., Ranka, S. & Sahni, S. *Strassen's matrix multiplication on GPUs in 2011 IEEE 17th International Conference on Parallel and Distributed Systems* (2011), 157 (cit. on p. 35).
115. Huang, J., Yu, C. D. & van de Geijn, R. A. *Implementing Strassen's Algorithm with CUTLASS on NVIDIA Volta GPUs. arXiv preprint arXiv:1808.07984* (2018) (cit. on p. 35).
116. D'Alberto, P. & Nicolau, A. *Using recursion to boost ATLAS's performance in High-Performance Computing* (2008), 142 (cit. on p. 35).
117. Fox, G. C., Otto, S. W. & Hey, A. J. *Matrix algorithms on a hypercube I: Matrix multiplication. Parallel computing*, 17 (1987) (cit. on p. 35).
118. Fox, G. C. *Solving problems on concurrent processors* (1988) (cit. on p. 35).
119. Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. & Whaley, R. C. *ScaLAPACK Users' Guide* (Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997) (cit. on pp. 35, 36, 108).
120. McColl, W. F. & Tiskin, A. *Memory-efficient matrix multiplication in the BSP model. Algorithmica* 24, 287 (1999) (cit. on p. 36).
121. Zheng, Q. & Lafferty, J. D. *Convergence Analysis for Rectangular Matrix Completion Using Burer-Monteiro Factorization and Gradient Descent in* (2016) (cit. on pp. 36, 81, 121).
122. Xiong, N. *Optimizing Tall-and-Skinny Matrix-Matrix Multiplication on GPUs* PhD thesis (UC Riverside, 2018) (cit. on p. 36).
123. Rogers, D. W. *Computational Chemistry Using the PC* (John Wiley & Sons, Inc., 2003) (cit. on p. 36).

124. Solcà, R., Kozhevnikov, A., Haidar, A., Tomov, S., Dongarra, J. & Schulthess, T. C. *Efficient Implementation of Quantum Materials Simulations on Distributed CPU-GPU Systems* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (ACM, Austin, Texas, 2015), 10:1. <http://doi.acm.org/10.1145/2807591.2807654> (cit. on p. 36).
125. Arge, L., Goodrich, M. T., Nelson, M. & Sitchinava, N. *Fundamental parallel algorithms for private-cache chip multiprocessors* in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures* (2008), 197 (cit. on pp. 37, 78).
126. Lee, H.-J., Robertson, J. P. & Fortes, J. A. *Generalized Cannon's algorithm for parallel matrix multiplication* in *Proceedings of the 11th international conference on Supercomputing* (1997), 44 (cit. on pp. 38, 79).
127. Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D. & Whaley, R. C. ScaLAPACK: A portable linear algebra library for distributed memory computers—Design issues and performance. *Computer Physics Communications* **97**, 1 (1996) (cit. on pp. 38, 81).
128. Sethi, R. Complete Register Allocation Problems. *SIAM journal on Computing*, 226 (1973) (cit. on pp. 39, 118, 146).
129. Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E. & Markstein, P. W. Register allocation via coloring. *Computer languages* **6**, 47 (1981) (cit. on p. 39).
130. Liu, Q. *Red-blue and standard pebble games: complexity and applications in the sequential and parallel models* PhD thesis (MIT, 2017) (cit. on pp. 43, 78, 95).
131. Loomis, L. H., Whitney, H., *et al.* An inequality related to the isoperimetric inequality. *Bulletin of the American Mathematical Society* **55**, 961 (1949) (cit. on pp. 52, 117).
132. Gerstenberger, R., Besta, M. & Hoefler, T. *Enabling highly-scalable remote memory access programming with MPI-3 one sided* in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013), 1 (cit. on p. 68).
133. Hoefler, T., Dinan, J., Thakur, R., Barrett, B., Balaji, P., Gropp, W. & Underwood, K. Remote memory access programming in MPI-3. *ACM Transactions on Parallel Computing (TOPC)* **2**, 1 (2015) (cit. on pp. 68, 106).

134. Vetter, J. S. & McCracken, M. O. Statistical scalability analysis of communication operations in distributed applications. *ACM SIGPLAN Notices*, 123 (2001) (cit. on p. 69).
135. Chan, S. M. *Just a pebble game in 2013 IEEE Conference on Computational Complexity* (2013), 133 (cit. on p. 78).
136. Wolf, M. E. & Lam, M. S. *A data locality optimizing algorithm in Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation* (1991), 30 (cit. on p. 79).
137. Kennedy, K. & McKinley, K. S. *Maximizing loop parallelism and improving data locality via loop fusion and distribution in LCPC* (1993), 301 (cit. on p. 79).
138. Darte, A. *On the complexity of loop fusion in PACT* (1999), 149 (cit. on pp. 79, 118, 147).
139. Toledo, S. A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms and Visualization*, 161 (1999) (cit. on p. 79).
140. Mohanty, S. K. I/O efficient algorithms for matrix computations. *arXiv preprint arXiv:1006.1307* (2010) (cit. on p. 79).
141. Scott, J., Holtz, O. & Schwartz, O. *Matrix multiplication I/O-complexity by path routing in Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures* (2015), 35 (cit. on p. 79).
142. Bender, M. A., Brodal, G. S., Fagerberg, R., Jacob, R. & Vicari, E. Optimal sparse matrix dense vector multiplication in the I/O-model. *Theory of Computing Systems* 47, 934 (2010) (cit. on p. 79).
143. Greiner, G. *Sparse matrix computations and their I/O complexity* PhD thesis (Technische Universität München, 2012) (cit. on p. 79).
144. Gupta, A. & Kumar, V. *Scalability of Parallel Algorithms for Matrix Multiplication in ICPP* (1993), 115 (cit. on p. 79).
145. Blumofe, R. D., Frigo, M., Joerg, C. F., Leiserson, C. E. & Randall, K. H. *An analysis of dag-consistent distributed shared-memory algorithms in Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures* (1996), 297 (cit. on p. 79).
146. Frigo, M., Leiserson, C. E., Prokop, H. & Ramachandran, S. *Cache-oblivious algorithms in 40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)* (1999), 285 (cit. on p. 79).

147. Lazzaro, A., VandeVondele, J., Hutter, J. & Schütt, O. *Increasing the efficiency of sparse matrix-matrix multiplication with a 2.5 D algorithm and one-sided MPI in Proceedings of the Platform for Advanced Scientific Computing Conference* (2017), 1 (cit. on p. 79).
148. Koanantakool, P., Azad, A., Buluç, A., Morozov, D., Oh, S.-Y., Olikier, L. & Yelick, K. *Communication-avoiding parallel sparse-dense matrix-matrix multiplication in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016), 842 (cit. on p. 79).
149. Kwasniewski, G., Kabic, M., Ben-Nun, T., Ziogas, A. N., Saethre, J. E., Gaillard, A., Schneider, T., Besta, M., Kozhevnikov, A., VandeVondele, J. & Hoefler, T. *On the Parallel I/O Optimality of Linear Algebra Kernels: Near-Optimal Matrix Factorizations in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Association for Computing Machinery, St. Louis, Missouri, 2021) (cit. on p. 81).
150. Krishnamoorthy, A. & Menon, D. *Matrix inversion using Cholesky decomposition in 2013 signal processing: Algorithms, architectures, arrangements, and applications (SPA)* (2013), 70 (cit. on pp. 81, 121).
151. Del Ben, M., Hutter, J. & VandeVondele, J. Electron correlation in the condensed phase from a resolution of identity approach based on the Gaussian and plane waves scheme. *Journal of chemical theory and computation*, 2654 (2013) (cit. on p. 81).
152. Dongarra, J. & Luszczek, P. in *Encyclopedia of Parallel Computing* (ed Padua, D.) 2055 (Springer US, Boston, MA, 2011). https://doi.org/10.1007/978-0-387-09766-4_157 (cit. on p. 81).
153. Kestor, G., Gioiosa, R., Kerbyson, D. J. & Hoisie, A. *Quantifying the energy cost of data movement in scientific applications in 2013 IEEE international symposium on workload characterization (IISWC)* (2013), 56 (cit. on pp. 81, 121).
154. Unat, D., Dubey, A., Hoefler, T., Shalf, J., Abraham, M., Bianco, M., Chamberlain, B. L., Cledat, R., Edwards, H. C., Finkel, H., Fuerlinger, K., Hannig, F., Jeannot, E., Kamil, A., Keasler, J., Kelly, P. H. J., Leung, V., Ltaief, H., Maruyama, N., Newburn, C. J. & Pericás, M. *Trends in Data Locality Abstractions for HPC Systems in* (2017), 3007 (cit. on pp. 81, 121).

155. Hutter, E. & Solomonik, E. *Communication-avoiding Cholesky-QR₂ for rectangular matrices* in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2019), 89 (cit. on pp. 81–83, 107, 108, 115, 118, 147).
156. Gates, M., Kurzak, J., Charara, A., YarKhan, A. & Dongarra, J. *SLATE: design of a modern distributed and accelerated linear algebra library* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2019), 1 (cit. on pp. 82, 83, 107, 108, 115, 118, 119).
157. Solomonik, E. *Provably efficient algorithms for numerical tensor algebra* PhD thesis (UC Berkeley, 2014) (cit. on pp. 82, 83, 107, 118).
158. Intel. *Math Kernel Library* 2020. <https://software.intel.com/en-us/mkl> (cit. on pp. 82, 83, 106, 107, 115, 118).
159. Solomonik, E. *Communication Avoiding Numerical Dense Matrix Computations* 2021. <https://github.com/solomonik/CANDMC> (cit. on pp. 83, 108).
160. Hutter, E. *Communication-Avoiding Parallelism-Increasing maTriX factorization Library* <https://github.com/huttered40/capital> (cit. on pp. 83, 108).
161. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A. & Sadayappan, P. in *Compiler Construction: 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings* (ed Hendren, L.) 132 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008). https://doi.org/10.1007/978-3-540-78791-4_9 (cit. on pp. 91, 147).
162. Olivry, A., Langou, J., Pouchet, L.-N., Sadayappan, P. & Rastello, F. *Automated derivation of parametric data movement lower bounds for affine programs* in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020), 808 (cit. on pp. 91, 99, 100, 117, 118, 121–123, 144, 145, 147).
163. Kuhn, H. W. & Tucker, A. W. *Nonlinear programming*, 247 (2014) (cit. on p. 92).
164. Alwen, J. & Serbinenko, V. *High parallel complexity graphs and memory-hard functions* in *Proceedings of the forty-seventh annual ACM symposium on Theory of computing* (2015), 595 (cit. on p. 96).

165. Karp, R. M. A survey of parallel algorithms for shared-memory machines (1988) (cit. on p. 96).
166. Dongarra, J., Faverge, M., Ltaief, H. & Luszczek, P. Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting. *Concurrency and Computation: Practice and Experience*, 1408 (2014) (cit. on pp. 101, 102).
167. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A. & Sorensen, D. *LAPACK Users' guide* (Siam, 1999) (cit. on p. 102).
168. Agullo, E., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Langou, J., Ltaief, H., Luszczek, P. & YarKhan, A. PLASMA Users' Guide. Parallel Linear Algebra Software for Multicore Architectures. *Rapport technique, Innovative Computing Laboratory, University of Tennessee* (2011) (cit. on p. 102).
169. Grigori, L., Demmel, J. W. & Xiang, H. *Communication avoiding Gaussian elimination in SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (2008), 1 (cit. on pp. 102, 104).
170. Quintana-Orti, G., Quintana-Orti, E. S., Geijn, R. A. V. D., Zee, F. G. V. & Chan, E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software (TOMS)*, 1 (2009) (cit. on p. 104).
171. Rabenseifner, R. & Träff, J. L. *More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems in European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting* (2004), 36 (cit. on p. 104).
172. Kabić, M., Pintarelli, S., Kozhevnikov, A. & VandeVondele, J. *COSTA: Communication-Optimal Shuffle and Transpose Algorithm with Process Relabeling in International Conference on High Performance Computing* (2021), 217 (cit. on p. 106).
173. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., *et al.* in *Tools for High Performance Computing 2011* 79 (Springer, 2012) (cit. on p. 107).
174. TOP500 list. *November 2019 TOP500 list* <https://www.top500.org/lists/2019/11> (April. 2020). 2020 (cit. on p. 108).

175. Ziogas, A. N., Ben-Nun, T., Fernández, G. I., Schneider, T., Luisier, M. & Hoefler, T. *A data-centric approach to extreme-scale ab initio dissipative quantum transport simulations* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2019), 1 (cit. on pp. 108, 116, 118).
176. Osawa, K., Tsuji, Y., Ueno, Y., Naruse, A., Yokota, R. & Matsuoka, S. *Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks* in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2019), 12359 (cit. on pp. 108, 116).
177. Bruno, J. & Sethi, R. Code generation for a one-register machine. *Journal of the ACM (JACM)*, 502 (1976) (cit. on p. 118).
178. Vitter, J. S. External memory algorithms, 1 (1998) (cit. on p. 117).
179. Feautrier, P. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International journal of parallel programming* 21, 313 (1992) (cit. on p. 117).
180. Invernizzi, A., Nikolov, T., Querciagrossa, L. & Solcà, R. *Distributed Linear Algebra with (HPX) Futures (forthcoming)* in *Proceedings of the Platform for Advanced Scientific Computing Conference* (2021) (cit. on p. 118).
181. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A. & Dongarra, J. *Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA* in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (2011), 1432 (cit. on p. 118).
182. Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S. & Tomov, S. in *GPU Computing Gems Jade Edition* 473 (Elsevier, 2012) (cit. on p. 118).
183. Poulson, J., Marker, B., Van de Geijn, R. A., Hammond, J. R. & Romero, N. A. *Elemental: A new framework for distributed memory dense matrix computations* in (ACM New York, NY, USA, 2013), 1 (cit. on p. 118).
184. Cray. *LibSci: Cray Scientific Libraries* 2020. https://olcf.ornl.gov/software_package/libsci/ (cit. on p. 118).
185. NVIDIA. *CUSOLVER Reference Guide* 2020. <https://docs.nvidia.com/cuda/cusolver> (cit. on p. 118).

186. Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Langou, J., Ltaief, H. & Tomov, S. *LU factorization for accelerator-based systems in 2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA) (2011)*, 217 (cit. on p. 118).
187. Tate, A., Kamil, A., Dubey, A., Größlinger, A., Chamberlain, B., Goglin, B., Edwards, C., Newburn, C. J., Padua, D., Unat, D., *et al.* *Programming abstractions for data locality in ()* (cit. on p. 121).
188. Ben-Nun, T. & Hoefler, T. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *ACM Comput. Surv.* (2019) (cit. on p. 121).
189. Solomonik, E., Matthews, D., Hammond, J. R., Stanton, J. F. & Demmel, J. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 3176 (2014) (cit. on p. 121).
190. Elango, V., Rastello, F., Pouchet, L.-N., Ramanujam, J. & Sadayappan, P. *On Characterizing the Data Access Complexity of Programs in Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (ACM, Mumbai, India, 2015)* (cit. on p. 121).
191. Demaine, E. D. & Liu, Q. C. *Red-Blue Pebble Game: Complexity of Computing the Trade-Off between Cache Size and Memory Transfers in Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (2018)*, 195 (cit. on pp. 121, 146).
192. Zhang, X., Xiao, J. & Tan, G. *I/O Lower Bounds for Auto-tuning of Convolutions in CNNs 2020* (cit. on pp. 121, 123, 144, 145, 147).
193. Pouchet, L. N. *PolyBench: The Polyhedral Benchmark suite 2016*. <https://sourceforge.net/projects/polybench> (cit. on p. 144).
194. Keasler, J. & USDOE. *Livermore Unstructured Lagrange Explicit Shock Hydrodynamics 2010*. <https://www.osti.gov/servlets/purl/1231396> (cit. on p. 146).
195. Baldauf, M., Seifert, A., Förstner, J., Majewski, D. & Raschendorfer, M. Operational convective-scale numerical weather prediction with the COSMO model: Description and sensitivities. *Monthly Weather Review*, 139:3387–3905 (2011) (cit. on p. 146).
196. Ivanov, A., Dryden, N., Ben-Nun, T., Li, S. & Hoefler, T. *Data Movement Is All You Need: A Case Study on Optimizing Transformers 2020* (cit. on p. 146).

197. LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* **86**, 2278 (1998) (cit. on p. 146).
198. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. & Polosukhin, I. *Attention Is All You Need* 2017 (cit. on p. 146).
199. Dymond, P. W. & Tompa, M. Speedups of deterministic machines by synchronous parallel machines. *Journal of Computer and System Sciences*, 149 (1985) (cit. on p. 146).
200. Savage, J. E. *Extending the Hong-Kung model to memory hierarchies in International Computing and Combinatorics Conference* (1995), 270 (cit. on p. 146).
201. Papp, P. A. & Wattenhofer, R. *On the Hardness of Red-Blue Pebble Games in Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (2020), 419 (cit. on p. 147).
202. Irony, D., Toledo, S. & Tiskin, A. *Communication lower bounds for distributed-memory matrix multiplication* 2004 (cit. on p. 147).
203. Besta, M. & Hoefler, T. *Accelerating irregular computations with hardware transactional memory and active messages in Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (2015), 161 (cit. on p. 147).
204. Sakr, S., Bonifati, A., Voigt, H., Iosup, A., Ammar, K., Angles, R., Aref, W., Arenas, M., Besta, M., Boncz, P. A., *et al.* The Future is Big Graphs! A Community View on Graph Processing Systems. *arXiv preprint arXiv:2012.06171* (2020) (cit. on p. 147).
205. Gianinazzi, L., Kalvoda, P., De Palma, A., Besta, M. & Hoefler, T. Communication-avoiding parallel minimum cuts and connected components. *ACM SIGPLAN Notices*, 219 (2018) (cit. on p. 147).
206. Besta, M., Fischer, M., Ben-Nun, T., Stanojevic, D., Licht, J. D. F. & Hoefler, T. *Substream-centric maximum matchings on fpga* in (ACM New York, NY, USA, 2020), 1 (cit. on p. 147).
207. Besta, M., Kanakagiri, R., Mustafa, H., Karasikov, M., Rättsch, G., Hoefler, T. & Solomonik, E. *Communication-efficient jaccard similarity for high-performance distributed genome comparisons in 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2020), 1122 (cit. on p. 147).

208. Besta, M., Podstawski, M., Groner, L., Solomonik, E. & Hoefler, T. *To push or to pull: On reducing communication and synchronization in graph computations* in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing* (2017), 93 (cit. on p. 147).
209. Besta, M., Vonarburg-Shmaria, Z., Schaffner, Y., Schwarz, L., Kwasniewski, G., Gianinazzi, L., Beranek, J., Janda, K., Hostenstein, T., Leisinger, S., *et al.* GraphMineSuite: Enabling High-Performance and Programmable Graph Mining Algorithms with Set Algebra. *arXiv preprint arXiv:2103.03653* (2021) (cit. on p. 147).
210. Chen, X., Han, Y. & Wang, Y. *Communication Lower Bound in Convolution Accelerators* in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), 529 (cit. on p. 147).
211. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A. & Sadayappan, P. *Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model* in *International Conference on Compiler Construction* (2008), 132 (cit. on p. 147).
212. Grosser, T., Groesslinger, A. & Lengauer, C. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 1250010 (2012) (cit. on p. 147).
213. Hoefler, T. & Kwasniewski, G. *Automatic complexity analysis of explicitly parallel programs* in *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures* (2014), 226 (cit. on p. 147).
214. Elango, V., Rastello, F., Pouchet, L.-N., Ramanujam, J. & Sadayappan, P. *On characterizing the data movement complexity of computational DAGs for parallel execution* in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (2014), 296 (cit. on p. 147).
215. Ballard, G., Buluc, A., Demmel, J., Grigori, L., Lipshitz, B., Schwartz, O. & Toledo, S. *Communication optimal parallel multiplication of sparse random matrices* in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures* (2013), 222 (cit. on p. 151).
216. Azad, A., Ballard, G., Buluc, A., Demmel, J., Grigori, L., Schwartz, O., Toledo, S. & Williams, S. *Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication*. *SIAM Journal on Scientific Computing*, C624 (2016) (cit. on p. 151).

217. Ballard, G., Druinsky, A., Knight, N. & Schwartz, O. Hypergraph partitioning for sparse matrix-matrix multiplication. *ACM Transactions on Parallel Computing (TOPC)*, 1 (2016) (cit. on p. 151).
218. De Fine Licht, J., Kwasniewski, G. & Hoefler, T. *Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis* in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2020), 244 (cit. on p. 151).

CURRICULUM VITAE

PERSONAL DATA

Name Grzegorz Kwasniewski
Date of Birth July 1, 1989
Place of Birth Krakow, Poland
Citizen of Poland

EDUCATION

02.2011–09.2012 AGH UST,
Krakow, Poland
Final degree: Master of Science in Automatics and
Robotics
10.2008–02.2011 AGH UST,
Krakow, Poland
Final degree: Bachelor of Science in Automatics and
Robotics

EMPLOYMENT

07–10.2021 PhD Intern
Huawei Research Center,
Munich, Germany
2014–2021 Scientific Assistant
SPCL lab,
Zurich, Switzerland
2013 Software Engineer
IBM,
Krakow, Poland
2011–2012 Software Engineer
Progress & Business Foundation,
Krakow, Poland